MALT : A MALloc Tracker

A memory profiling tool

DE LA RECHERCHE À L'INDUSTRIE
cea

GENCI
GRAND EQUIPEMENT NATIONAL POUR LE CALCUL INTENSIF

intel

UNIVERSITÉ DE
VERSAILLES
ST-QUENTIN-EN-YVELINES

THE QUESTION

Goal

- We have **profiling tool** for **timing** (eg. Valgrind or vtune)

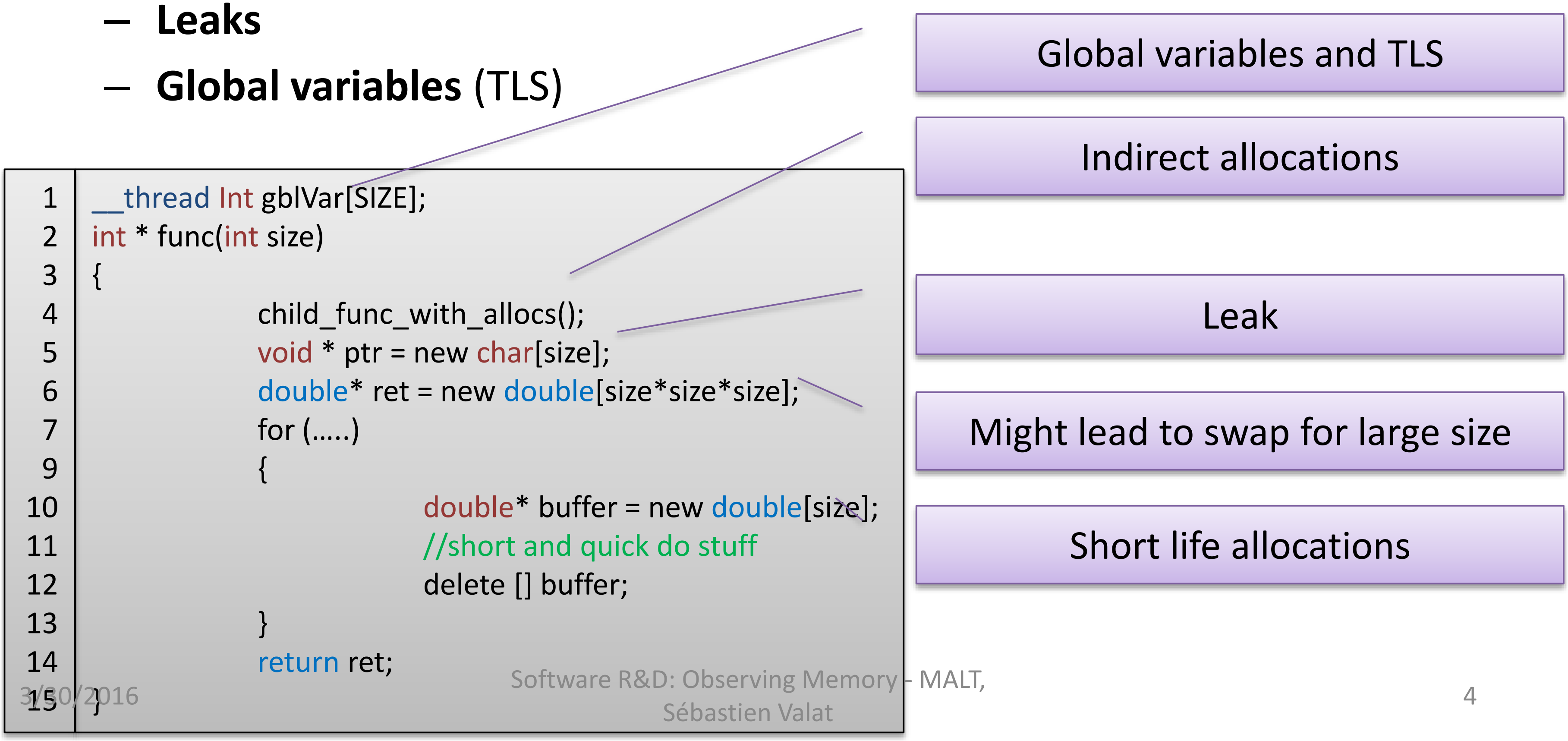
- But for **memory usage** ?

- Memory can be an issue :
  - Failed to run (or swap) due to **lack of memory resource**.
  - **Performance impact** of memory management functions

- Three main questions :
  - How to reduce **memory footprint** ?
  - How to improve overhead of **memory management** ?
  - How to improve **memory usage** ?

Some issue examples

- We want to help searching :
  - **Where** memory is allocated.
  - **Properties** of allocated chunks.
  - **Bad** allocation **patterns** for performance.
  - **Leaks**
  - **Global variables** (TLS)



```
__thread Int gblVar[SIZE];
int * func(int size)
{
        child_func_with_allocs();
        void * ptr = new char[size];
        double* ret = new double[size*size*size];
        for (.....)
        {
                double* buffer = new double[size];
                //short and quick do stuff
                delete [] buffer;
        }
        return ret;
}
```

Global variables and TLS

Indirect allocations

Leak

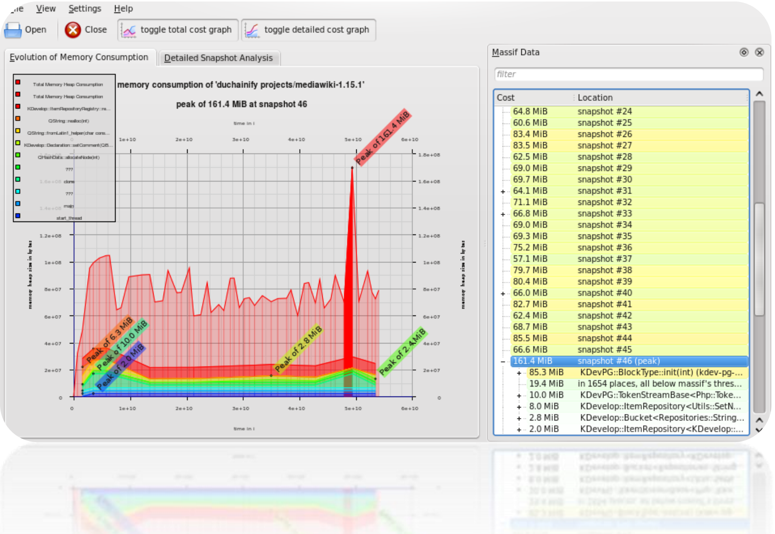
Might lead to swap for large size

Short life allocations

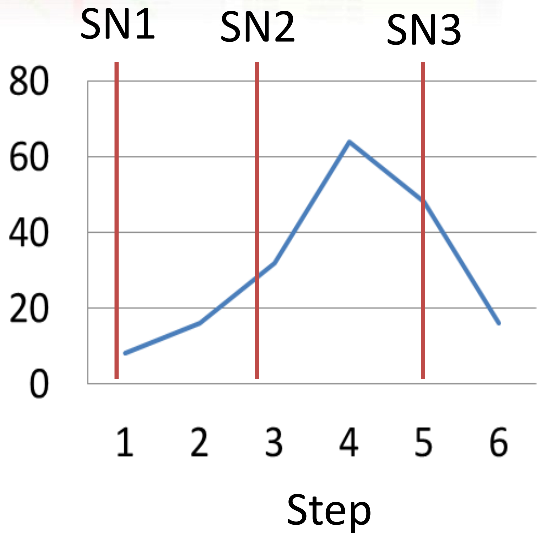
EXISTING TOOLS

Existing tools

- **Valgrind - massif :**
  - Take **snapshots** over time.
  - **Link memory** size to **functions**
  - **Peak** might **not be captured**.
  - Might **miss short live** allocations
  - GUI not adequate for large code
  - Slow, not parallel.


File
View
Settings
Help
Open
Close
toggle total cost graph
toggle detailed cost graph
Evolution of Memory Consumption
Detailed Snapshot Analysis
memory consumption of 'duchainify projects/mediawiki-1.15.1'
peak of 161.4 MiB at snapshot 46
Peak of 161.4 MiB
Peak of 6.3 MiB
Peak of 10.0 MiB
Peak of 2.0 MiB
Peak of 2.8 MiB
Peak of 2.4 MiB
time in i
memory heap size in bytes
Massif Data
filter

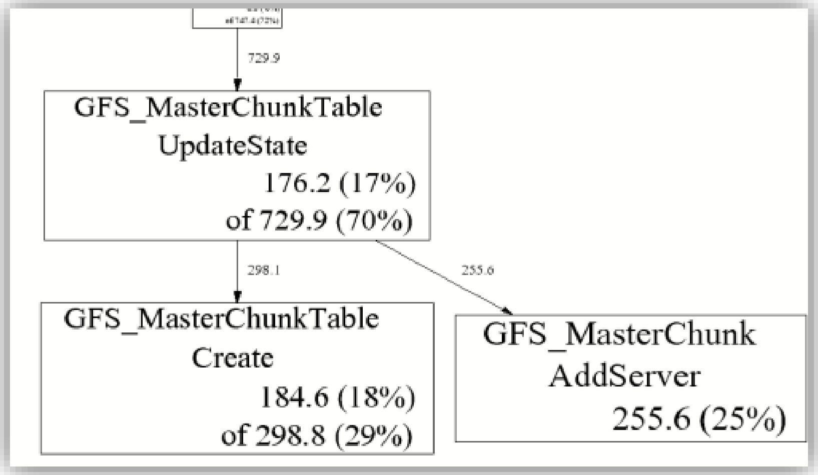
Cost	Location
64.8 MiB	snapshot #24
60.6 MiB	snapshot #25
83.4 MiB	snapshot #26
83.5 MiB	snapshot #27
62.5 MiB	snapshot #28
69.0 MiB	snapshot #29
69.7 MiB	snapshot #30
64.1 MiB	snapshot #31
71.1 MiB	snapshot #32
66.8 MiB	snapshot #33
69.0 MiB	snapshot #34
69.3 MiB	snapshot #35
75.2 MiB	snapshot #36
57.1 MiB	snapshot #37
79.7 MiB	snapshot #38
80.4 MiB	snapshot #39
66.0 MiB	snapshot #40
82.7 MiB	snapshot #41
62.4 MiB	snapshot #42
68.7 MiB	snapshot #43
85.5 MiB	snapshot #44
66.6 MiB	snapshot #45
161.4 MiB	snapshot #46 (peak)
85.3 MiB	KDevPG::BlockType::init(int) (kdev-pg-...
19.4 MiB	in 1654 places, all below massif's thres...
10.0 MiB	KDevPG::TokenStreamBase<Php::Toke...
8.0 MiB	KDevelop::ItemRepository<Utils::SetN...
2.8 MiB	KDevelop::Bucket<Repositories::String...
2.0 MiB	KDevelop::ItemRepository<KDevelop::...



- **Valgrind - memcheck :**
  - Misuse of memory functions (malloc/new/free….)
  - Leak detection
  - Invalid accesses
  - Slow, not parallel.

Existing tools

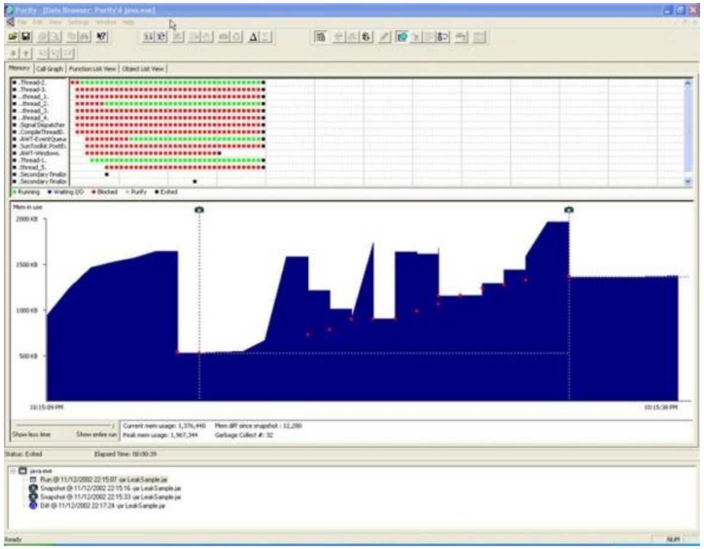
- **Google heap profiler (tcmalloc):**
  - Small overhead.
  - Only provide snapshots of allocated memory per stacks.
  - Peak might not be captured.
  - No binary or source instrumentation.
  - Output is not clear…
  - Lack of a real GUI to use it.



```
% pprof gfs_master profile.0100.heap
  255.6  24.7%  24.7%    255.6  24.7% GFS_MasterChunk::AddServer
  184.6  17.8%  42.5%    298.8  28.8% GFS_MasterChunkTable::Create
  176.2  17.0%  59.5%    729.9  70.5% GFS_MasterChunkTable::UpdateState
  169.8  16.4%  75.9%    169.8  16.4% PendingClone::PendingClone
   76.3   7.4%  83.3%     76.3   7.4% __default_alloc_template::_S_chunk_alloc
   49.5   4.8%  88.0%     49.5   4.8% hashtable::resize
```

Existing tools

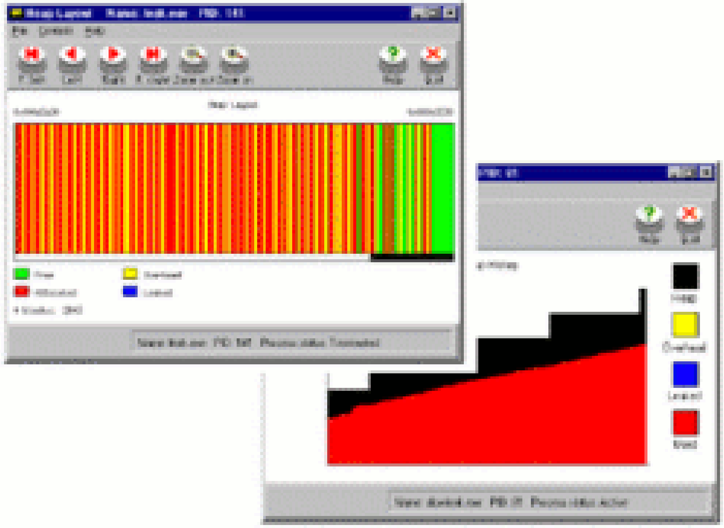
- **IBM Purify++ / Parasoft Insure++**
  - Commercial
  - Leak detection, access checking, memory debugging tools.
  - Use binary or source instrumentation.
  - Windows / Redhat
  - Quite old GUI



Current mem usage: 1,376,440 Mem diff since snapshot : 12,200
Peak mem usage: 1,967,344 Garbage Collect #: 32

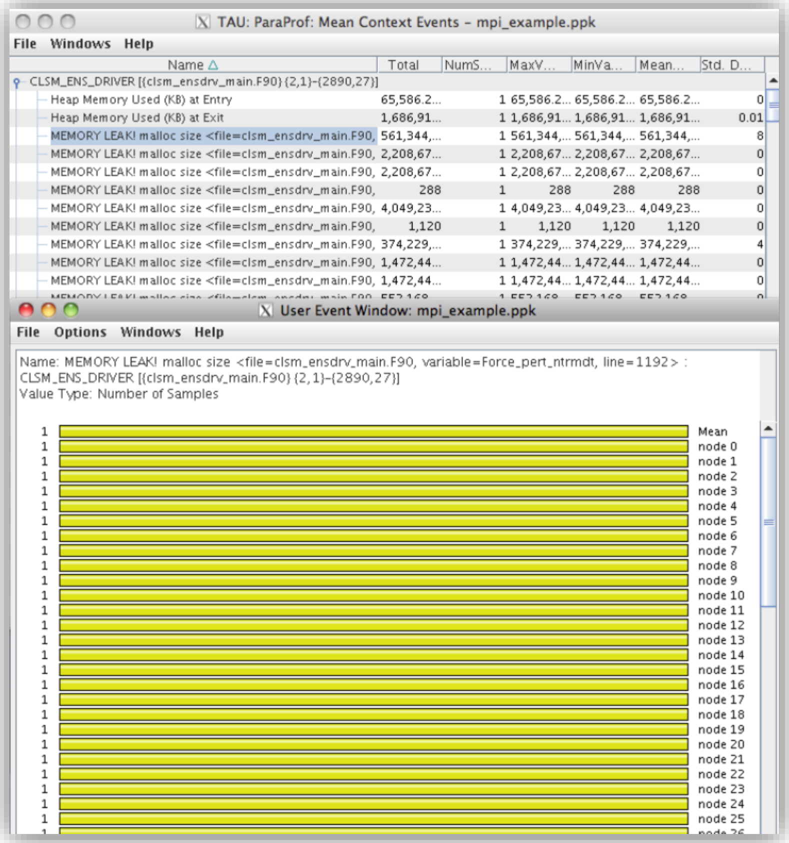
Status: Exited Elapsed Time: 00:00:39

java.exe
Run @ 11/12/2002 22:15:07 -jar LeakSample.jar
Snapshot @ 11/12/2002 22:15:16 -jar LeakSample.jar
Snapshot @ 11/12/2002 22:15:33 -jar LeakSample.jar
Diff @ 11/12/2002 22:17:24 -jar LeakSample.jar



Existing tools

- **TAU memory profiler**
  - Provide profiles
  - Follow stacks
  - Track leaks
  - Parallel, done for HPC/MPI
  - Lack easy matching with sources



TAU: ParaProf: Mean Context Events – mpi_example.ppk

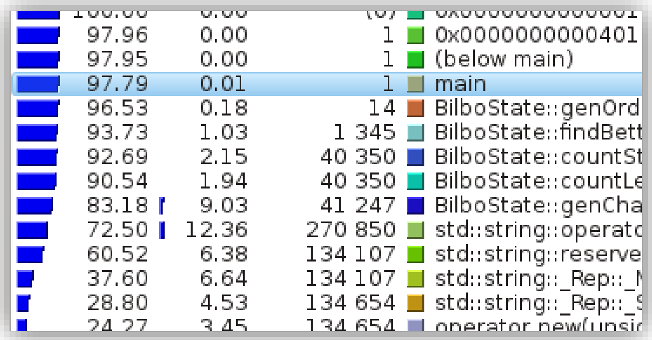
Name △	Total	NumS...	MaxV...	MinVa...	Mean...	Std. D...
CLSM_ENS_DRIVER [{clsm_ensdrv_main.F90} {2,1}-{2890,27}]						
Heap Memory Used (KB) at Entry	65,586.2...	1	65,586.2...	65,586.2...	65,586.2...	0
Heap Memory Used (KB) at Exit	1,686,91...	1	1,686,91...	1,686,91...	1,686,91...	0.01
MEMORY LEAK! malloc size <file=clsm_ensdrv_main.F90,	561,344,...	1	561,344,...	561,344,...	561,344,...	8
MEMORY LEAK! malloc size <file=clsm_ensdrv_main.F90,	2,208,67...	1	2,208,67...	2,208,67...	2,208,67...	0
MEMORY LEAK! malloc size <file=clsm_ensdrv_main.F90,	2,208,67...	1	2,208,67...	2,208,67...	2,208,67...	0
MEMORY LEAK! malloc size <file=clsm_ensdrv_main.F90,	288	1	288	288	288	0
MEMORY LEAK! malloc size <file=clsm_ensdrv_main.F90,	4,049,23...	1	4,049,23...	4,049,23...	4,049,23...	0
MEMORY LEAK! malloc size <file=clsm_ensdrv_main.F90,	1,120	1	1,120	1,120	1,120	0
MEMORY LEAK! malloc size <file=clsm_ensdrv_main.F90,	374,229,...	1	374,229,...	374,229,...	374,229,...	4
MEMORY LEAK! malloc size <file=clsm_ensdrv_main.F90,	1,472,44...	1	1,472,44...	1,472,44...	1,472,44...	0
MEMORY LEAK! malloc size <file=clsm_ensdrv_main.F90,	1,472,44...	1	1,472,44...	1,472,44...	1,472,44...	0
MEMORY LEAK! malloc size <file=clsm_ensdrv_main.F90,	553,168	1	553,168	553,168	553,168	0

User Event Window: mpi_example.ppk

Name: MEMORY LEAK! malloc size <file=clsm_ensdrv_main.F90, variable=Force_pert_ntrmdt, line=1192> :
CLSM_ENS_DRIVER [{clsm_ensdrv_main.F90} {2,1}-{2890,27}]
Value Type: Number of Samples

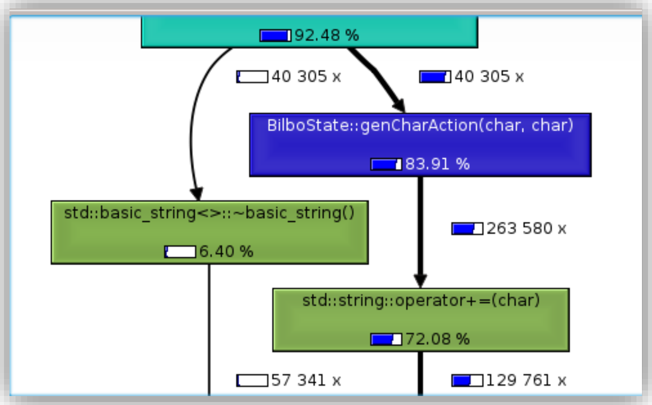
What is good in kcachgrind

- List of **functions** with **exclusive/inclusive** costs

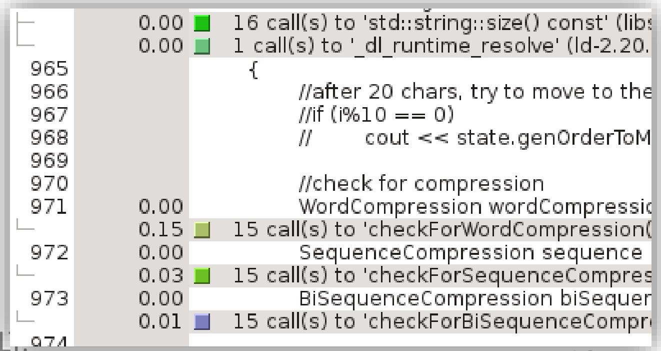


```
100.00   0.00        (0) 0x000000000000001
 97.96   0.00          1 0x0000000000401
 97.95   0.00          1 (below main)
 97.79   0.01          1 main
 96.53   0.18         14 BilboState::genOrd
 93.73   1.03      1 345 BilboState::findBett
 92.69   2.15     40 350 BilboState::countSt
 90.54   1.94     40 350 BilboState::countLe
 83.18   9.03     41 247 BilboState::genCha
 72.50  12.36    270 850 std::string::operato
 60.52   6.38    134 107 std::string::reserve
 37.60   6.64    134 107 std::string::_Rep::_I
 28.80   4.53    134 654 std::string::_Rep::_S
 24.27   3.45    134 654 operator new(unsic
```

- Nice **call tree**


92.48 %
40 305 x
40 305 x
BilboState::genCharAction(char, char)
83.91 %
std::basic_string<>::~basic_string()
6.40 %
263 580 x
std::string::operator+=(char)
72.08 %
57 341 x
129 761 x

- **Annotated** sources



```
        0.00    16 call(s) to 'std::string::size() const' (libs
        0.00    1 call(s) to '_dl_runtime_resolve' (ld-2.20.
 965              {
 966                     //after 20 chars, try to move to the
 967                     //if (i%10 == 0)
 968                     //       cout << state.genOrderToM
 969
 970                     //check for compression
 971    0.00             WordCompression wordCompressio
        0.15    15 call(s) to 'checkForWordCompression(
 972    0.00             SequenceCompression sequence
        0.03    15 call(s) to 'checkForSequenceCompres
 973    0.00             BiSequenceCompression biSequen
        0.01    15 call(s) to 'checkForBiSequenceCompre
 974
```

WHERE I GO

What I want to provide

- Same **approach** than **valgrind/kcachgind**

- **Mapped** allocations on **sources lines**

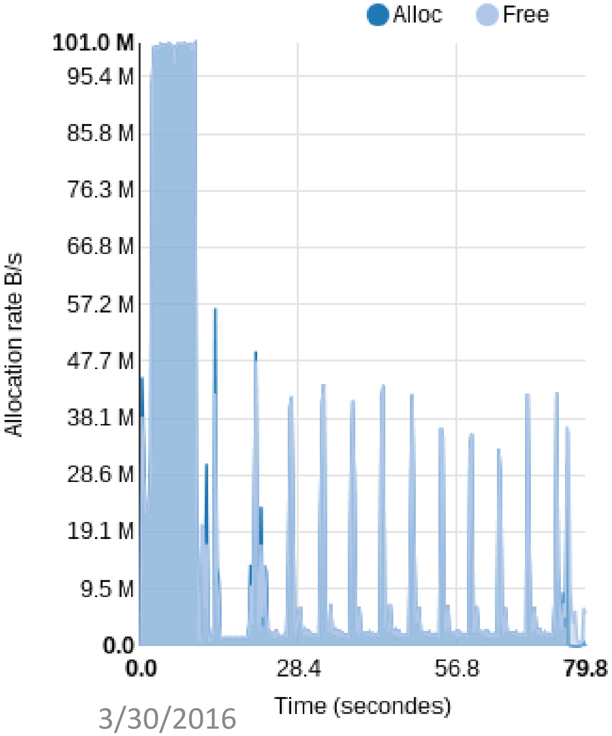
- For **memory resource usage** :
  - Memory **leaks** (malloc without free)
  - **Peak** and **total** allocated **memory**

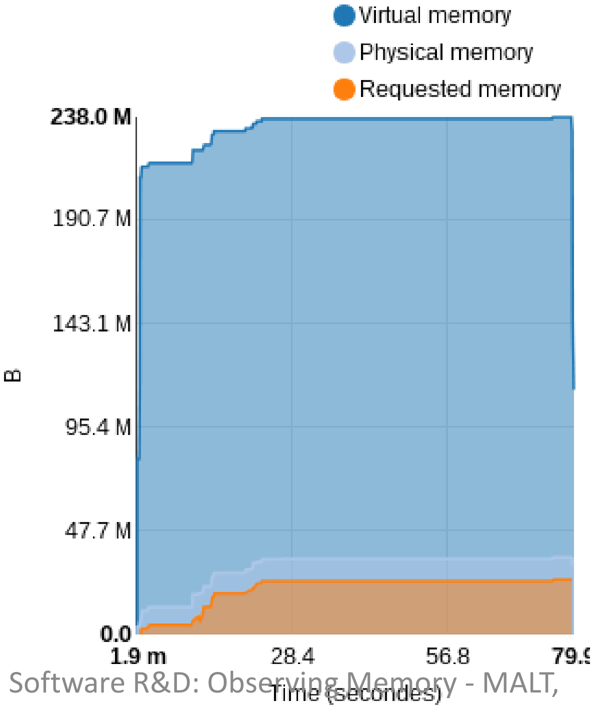
- For **performance** :
  - Allocation **count**
  - Allocation **sizes** (min/mean/max)
  - Chunk **lifetime** (min/mean/max)

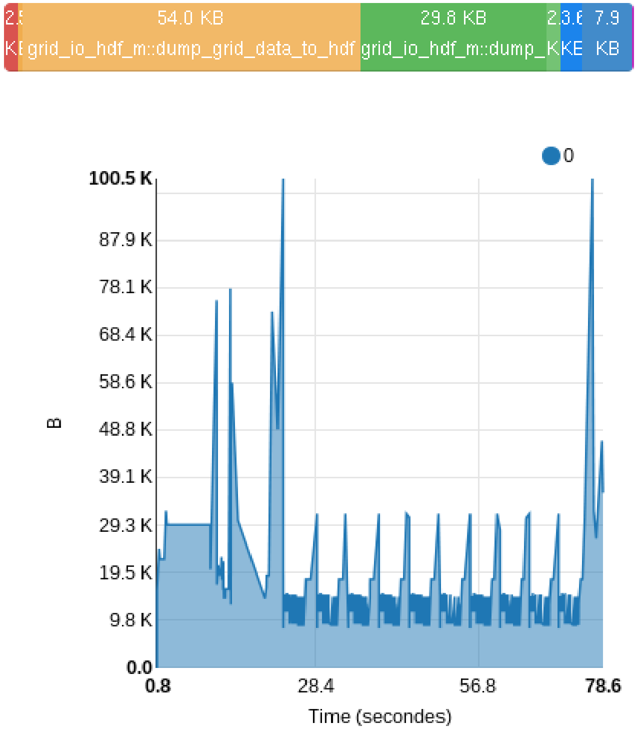
Temporal metrics

- **Profile over time :**
  - Allocation **rate**
  - **Physical / Virtual / Requested** memory
  - **Stack size** for each **thread** (require function instrumentation)

- **Example on YALES2 with gfortran :**

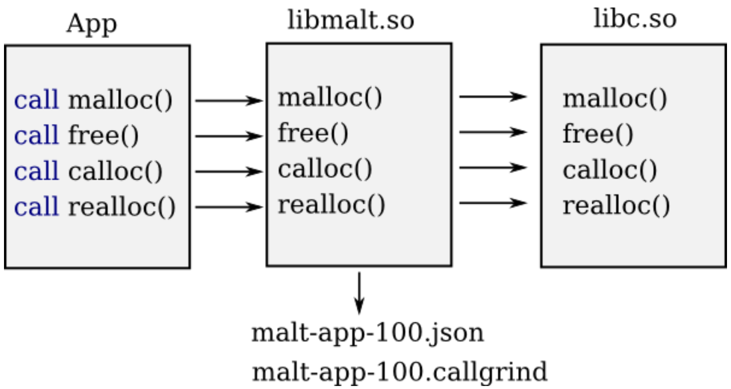






How it works

- Use LD_PRELOAD to intercept malloc/free/…



- Profile allocations on call stacks
- Generate JSON output file
- Build profile so size is limited by call tree

Tracking stacks

- Two approach implemented : backtrace and instrumentation

- Backtrace (default) :
  - No change on binary or sources.
  - Function from glibc
  - Manage all dynamic libraries
  - No impact on compute
  - Slow on x86_64 due to use of libunwind.
  - Slow for large number of calls (~>10M)

- Instrumentation :
  - Need source recompilation (available) : *-finstrument-function*
  - Or tools for binary instrumentation : MAQAO / Pintool (experimental)
  - Impact performance of compute only functioans
  - Faster for really large number of calls to malloc
  - Only provide stacks for the instrumented sources/binaries

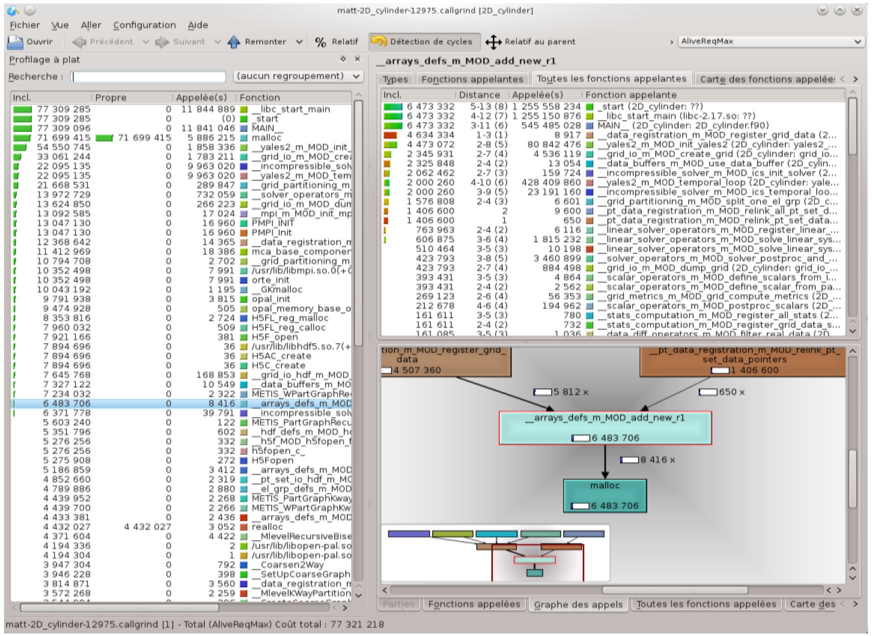
Global output

Callgrind compatibiltiy

- Can use kcachgrind
- Might be usefull for some users, cannot provide all metrics.

Own web view

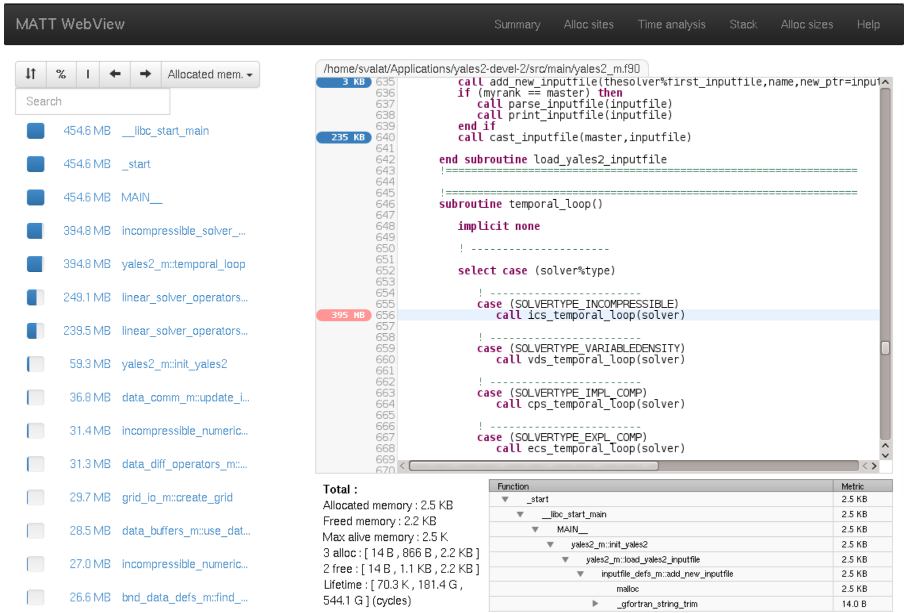
- Get all metrics
- Web technology (NodeJS, D3JS, Jquery, AngularJS)
- Easier for remote usage
- Can be used for shared working



Incl.	Propre	Appelée(s)	Fonction
77 309 285	0	11 844 889	__libc_start_main
77 309 285	0	(0)	_start
77 309 096	0	11 841 046	MAIN__
71 699 415	71 699 415	5 886 215	malloc
54 550 745	0	1 858 336	__yales2_m_MOD_init_
33 061 244	0	1 783 211	__grid_io_m_MOD_crea
22 095 135	0	9 963 020	__incompressible_solv
22 095 135	0	9 963 020	__yales2_m_MOD_tem
21 668 531	0	289 847	__grid_partitioning_m_
13 972 729	0	732 059	__solver_operators_m
13 624 850	0	266 223	__grid_io_m_MOD_dum
13 092 585	0	17 024	__mpi_m_MOD_init_mp
13 047 130	0	16 960	PMPI_INIT
13 047 130	0	16 960	PMPI_Init
12 368 642	0	14 365	__data_registration_m
11 412 969	0	18 386	mca_base_componer
10 794 708	0	2 702	__grid_partitioning_m_
10 352 498	0	7 991	/usr/lib/libmpi.so.0(+0
10 352 498	0	7 991	orte_init
10 043 192	0	1 195	__GKmalloc
9 791 938	0	3 815	opal_init
9 474 928	0	505	opal_memory_base_o
8 353 816	0	2 724	H5FL_reg_malloc
7 960 032	0	509	H5FL_reg_calloc
7 921 166	0	381	H5F_open
7 894 696	0	36	/usr/lib/libhdf5.so.7(+
7 894 696	0	36	H5AC_create
7 894 696	0	36	H5C_create
7 645 768	0	168 853	__grid_io_hdf_m_MOD_
7 327 122	0	10 549	__data_buffers_m_MO
7 234 032	0	2 322	METIS_WPartGraphRe
6 483 706	0	8 416	__arrays_defs_m_MOD
6 371 778	0	39 791	__incompressible_solv
5 603 240	0	122	METIS_PartGraphRecu
5 351 796	0	602	__hdf_defs_m_MOD_hc
5 276 256	0	332	__h5f_MOD_h5fopen_f
5 276 256	0	332	h5fopen_c_
5 275 908	0	272	H5Fopen
5 186 859	0	3 412	__arrays_defs_m_MOD
4 852 660	0	2 319	__pt_set_io_hdf_m_MC
4 789 886	0	2 880	__el_grp_defs_m_MOD
4 439 952	0	2 268	METIS_PartGraphKway
4 439 700	0	2 266	METIS_WPartGraphKw
4 433 381	0	2 436	__arrays_defs_m_MOD
4 432 027	4 432 027	3 052	realloc
4 371 604	0	4 422	__MlevelRecursiveBise
4 194 336	0	2	/usr/lib/libopen-pal.so
4 194 304	0	1	/usr/lib/libopen-pal.so
3 947 304	0	792	__Coarsen2Way
3 946 228	0	398	__SetUpCoarseGraph
3 814 871	0	3 560	__data_registration_m
3 572 268	0	2 259	__MlevelKWayPartition

Incl.	Distance	Appelée(s)	Fonction appelante
6 473 332	5-13 (8)	1 255 558 234	_start (2D_cylinder: ??)
6 473 332	4-12 (7)	1 255 150 876	__libc_start_main (libc-2.17.so: ??)
6 473 332	3-11 (6)	545 485 028	MAIN__ (2D_cylinder: 2D_cylinder.f90)
4 634 334	1-3 (1)	8 917	__data_registration_m_MOD_register_grid_data (2...
4 473 072	2-8 (5)	80 842 476	__yales2_m_MOD_init_yales2 (2D_cylinder: yales2_...
2 345 931	2-7 (4)	4 536 119	__grid_io_m_MOD_create_grid (2D_cylinder: grid_io...
2 325 848	2-4 (2)	13 054	__data_buffers_m_MOD_use_data_buffer (2D_cylin...
2 062 462	2-7 (3)	159 724	__incompressible_solver_m_MOD_ics_init_solver (2...
2 000 260	4-10 (6)	428 409 860	__yales2_m_MOD_temporal_loop (2D_cylinder: yale...
2 000 260	3-9 (5)	23 191 160	__incompressible_solver_m_MOD_ics_temporal_loo...
1 576 808	2-4 (3)	6 601	__grid_partitioning_m_MOD_split_one_el_grp (2D_c...
1 406 600	2	9 600	__pt_data_registration_m_MOD_relink_all_pt_set_d...
1 406 600	1	650	__pt_data_registration_m_MOD_relink_pt_set_data...
763 963	2-4 (2)	6 116	__linear_solver_operators_m_MOD_register_linear_...
606 875	3-6 (4)	1 815 232	__linear_solver_operators_m_MOD_solve_linear_sys...
510 464	3-5 (3)	10 198	__linear_solver_operators_m_MOD_solve_linear_sys...
423 793	3-8 (5)	3 460 899	__solver_operators_m_MOD_solver_postproc_and_...
423 793	2-7 (4)	884 498	__grid_io_m_MOD_dump_grid (2D_cylinder: grid_io_...
393 431	3-5 (3)	4 864	__scalar_operators_m_MOD_define_scalars_from_i...
393 431	2-4 (2)	2 562	__scalar_operators_m_MOD_define_scalar_from_pa...
269 123	2-6 (4)	56 353	__grid_metrics_m_MOD_grid_compute_metrics (2D_...
212 678	4-6 (4)	194 962	__scalar_operators_m_MOD_postproc_scalars (2D_...
161 611	3-5 (3)	780	__stats_computation_m_MOD_register_all_stats (2...
161 611	2-4 (2)	732	__stats_computation_m_MOD_register_grid_data_s...
161 085	3-5 (3)	1 036	__data_diff_operators_m_MOD_filter_real_data (2D...

matt-2D_cylinder-12975.callgrind [1] - Total (AliveReqMax) Coût total : 77 321 218



454.6 MB	__libc_start_main
454.6 MB	_start
454.6 MB	MAIN__
394.8 MB	incompressible_solver_...
394.8 MB	yales2_m::temporal_loop
249.1 MB	linear_solver_operators...
239.5 MB	linear_solver_operators...
59.3 MB	yales2_m::init_yales2
36.8 MB	data_comm_m::update_i...
31.4 MB	incompressible_numeric...
31.3 MB	data_diff_operators_m::...
29.7 MB	grid_io_m::create_grid
28.5 MB	data_buffers_m::use_dat...
27.0 MB	incompressible_numeric...
26.6 MB	bnd_data_defs_m::find_...

/home/svalat/Applications/yales2-devel-2/src/main/yales2_m.f90

```
      call add_new_inputfile(thesolver%first_inputfile,name,new_ptr=input
      if (myrank == master) then
         call parse_inputfile(inputfile)
         call print_inputfile(inputfile)
      end if
      call cast_inputfile(master,inputfile)

   end subroutine load_yales2_inputfile
   !===================================================================

   !===================================================================
   subroutine temporal_loop()

      implicit none

      ! ---------------------

      select case (solver%type)

         ! -----------------------
         case (SOLVERTYPE_INCOMPRESSIBLE)
            call ics_temporal_loop(solver)

         ! -----------------------
         case (SOLVERTYPE_VARIABLEDENSITY)
            call vds_temporal_loop(solver)

         ! -----------------------
         case (SOLVERTYPE_IMPL_COMP)
            call cps_temporal_loop(solver)

         ! -----------------------
         case (SOLVERTYPE_EXPL_COMP)
            call ecs_temporal_loop(solver)
```

Total :
Allocated memory : 2.5 KB
Freed memory : 2.2 KB
Max alive memory : 2.5 K
3 alloc : [14 B , 866 B , 2.2 KB]
2 free : [14 B , 1.1 KB , 2.2 KB]
Lifetime : [70.3 K , 181.4 G , 544.1 G] (cycles)

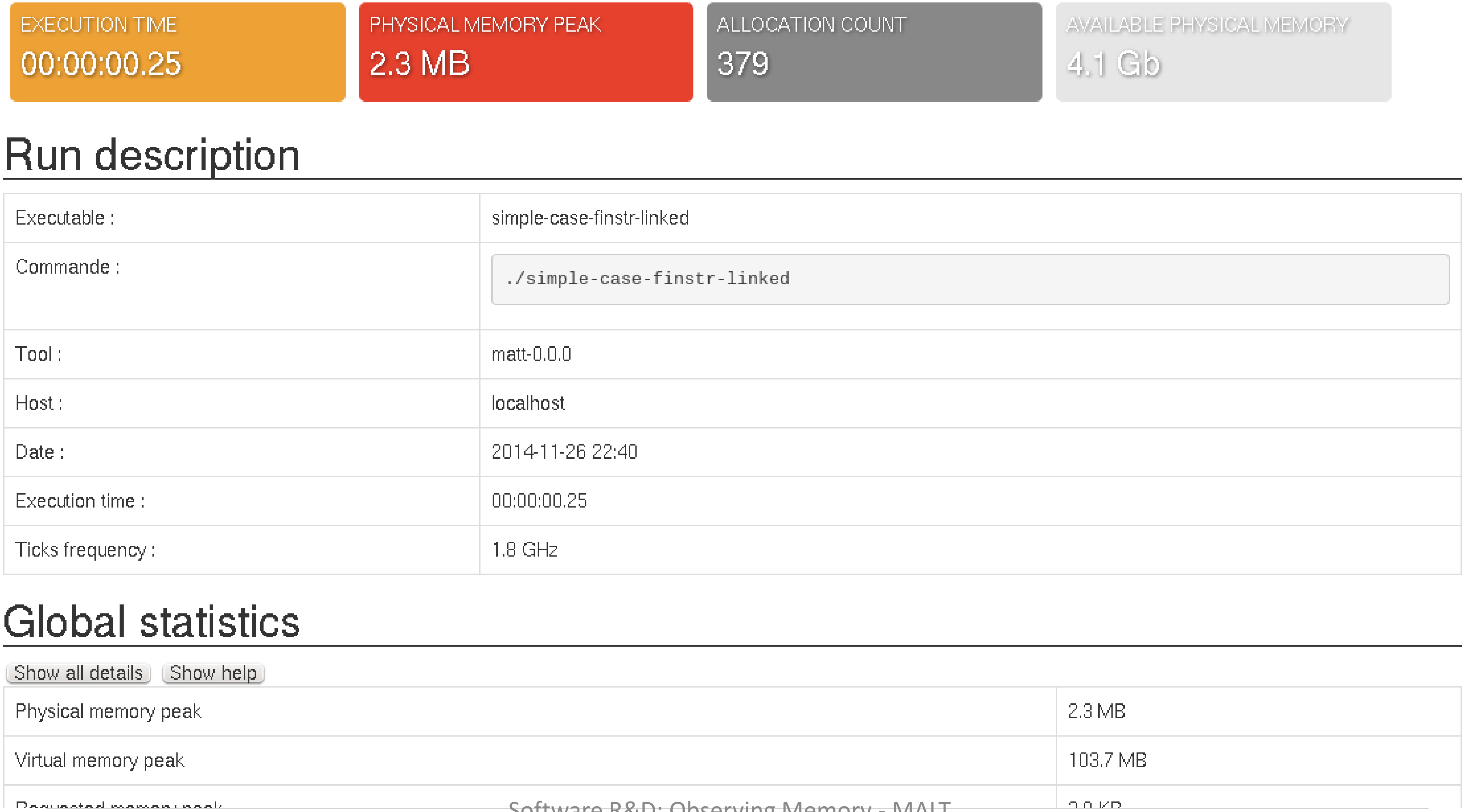
Function	Metric
▼ _start	2.5 KB
▼ __libc_start_main	2.5 KB
▼ MAIN__	2.5 KB
▼ yales2_m::init_yales2	2.5 KB
▼ yales2_m::load_yales2_inputfile	2.5 KB
▼ inputfile_defs_m::add_new_inputfile	2.5 KB
malloc	2.5 KB
▶ _gfortran_string_trim	14.0 B

What is missing to kcachegrind

- Display **human readable** units
  - You prefer **15728640** of **15MB** ?
  - I want to **compare to what I expect**.
- Cannot handle **non sum cumulative metrics**
  - **Inclusive** costs **only rely** on **+ operator**
  - Some mem. metrics **requires max/min** (eg. local peaks, lifetime, sizes)
- No way to express **time profiles**
- No way to express **parameter distributions** (eg. sizes).

SOME VIEWS

Global summary



EXECUTION TIME
00:00:00.25

PHYSICAL MEMORY PEAK
2.3 MB

ALLOCATION COUNT
379

AVAILABLE PHYSICAL MEMORY
4.1 Gb

Run description

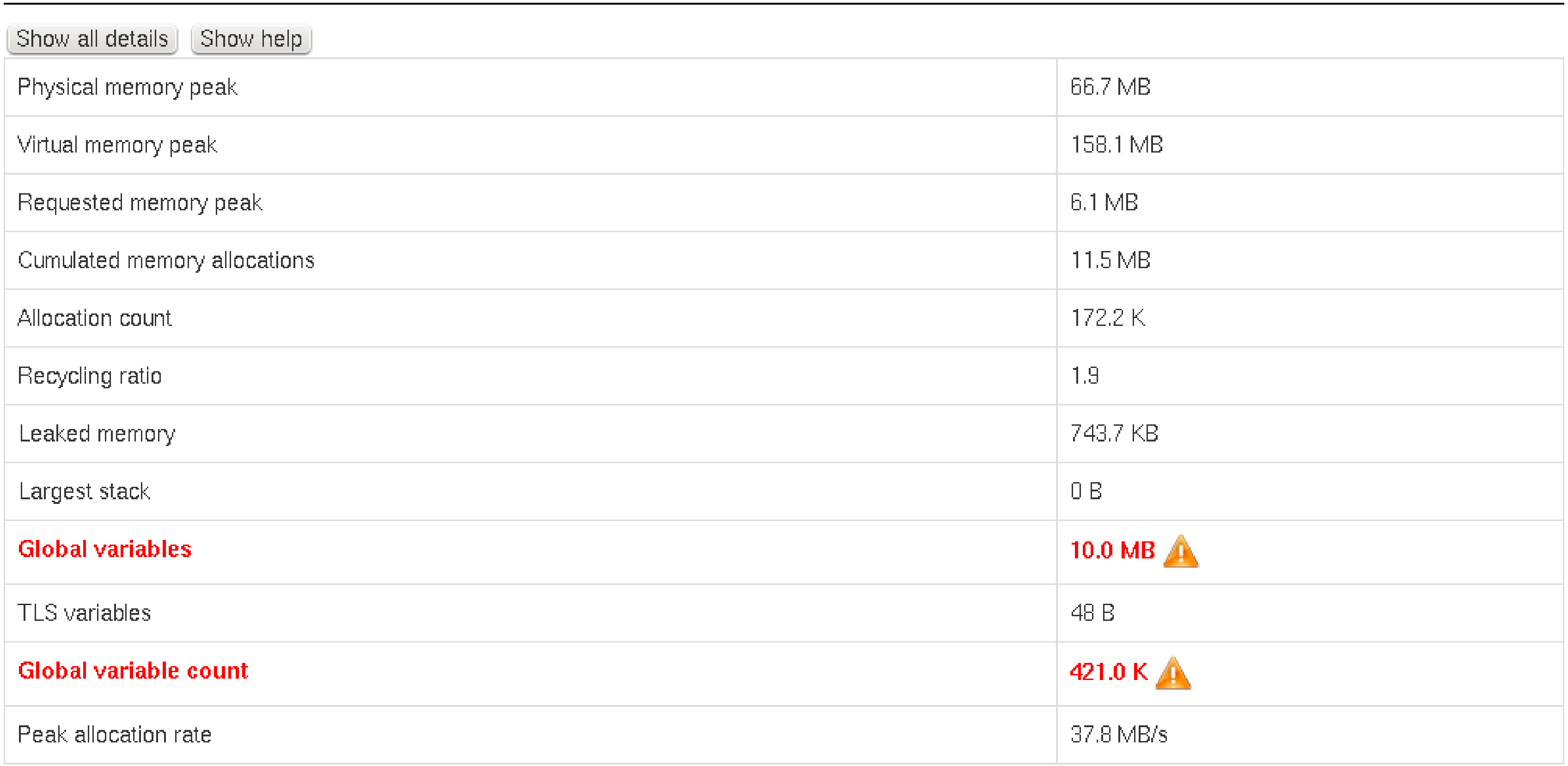
Executable :	simple-case-finstr-linked
Commande :	`./simple-case-finstr-linked`
Tool :	matt-0.0.0
Host :	localhost
Date :	2014-11-26 22:40
Execution time :	00:00:00.25
Ticks frequency :	1.8 GHz

Global statistics

Physical memory peak	2.3 MB
Virtual memory peak	103.7 MB
Requested memory peak	2.0 KB

Global summary

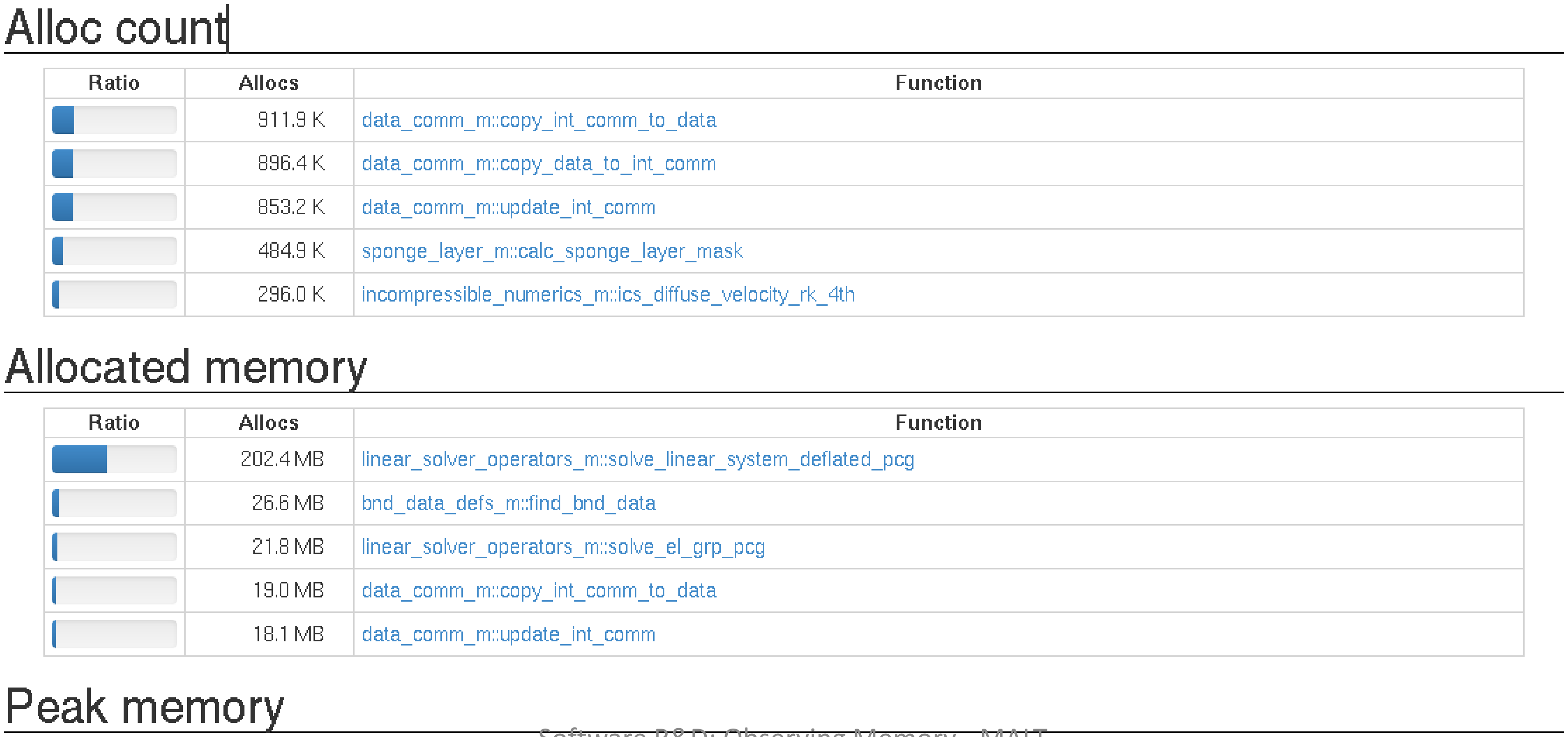
- Provide a small summary
- Provide some warnings



Physical memory peak	66.7 MB
Virtual memory peak	158.1 MB
Requested memory peak	6.1 MB
Cumulated memory allocations	11.5 MB
Allocation count	172.2 K
Recycling ratio	1.9
Leaked memory	743.7 KB
Largest stack	0 B
Global variables	**10.0 MB** ⚠
TLS variables	48 B
Global variable count	**421.0 K** ⚠
Peak allocation rate	37.8 MB/s

Global summary : top 5 functions

- Summarize **top functions** for some metrics
- Points to check
- Examples on YALES2



Alloc count

Ratio	Allocs	Function
	911.9 K	data_comm_m::copy_int_comm_to_data
	896.4 K	data_comm_m::copy_data_to_int_comm
	853.2 K	data_comm_m::update_int_comm
	484.9 K	sponge_layer_m::calc_sponge_layer_mask
	296.0 K	incompressible_numerics_m::ics_diffuse_velocity_rk_4th

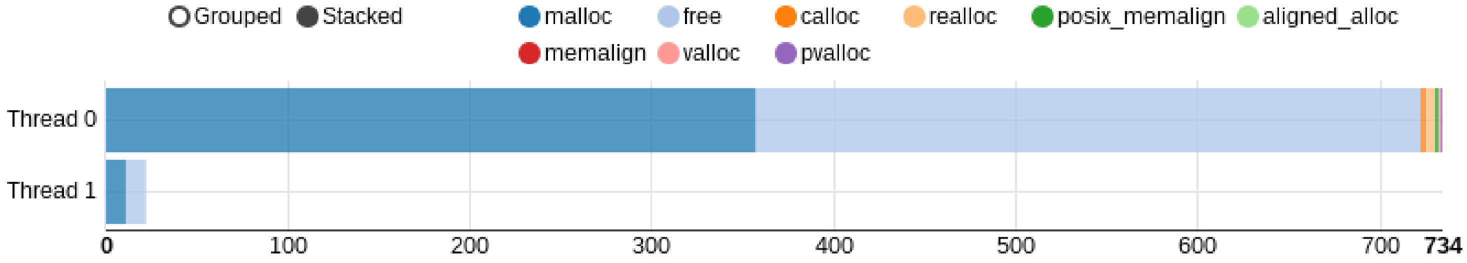
Allocated memory

Ratio	Allocs	Function
	202.4 MB	linear_solver_operators_m::solve_linear_system_deflated_pcg
	26.6 MB	bnd_data_defs_m::find_bnd_data
	21.8 MB	linear_solver_operators_m::solve_el_grp_pcg
	19.0 MB	data_comm_m::copy_int_comm_to_data
	18.1 MB	data_comm_m::update_int_comm

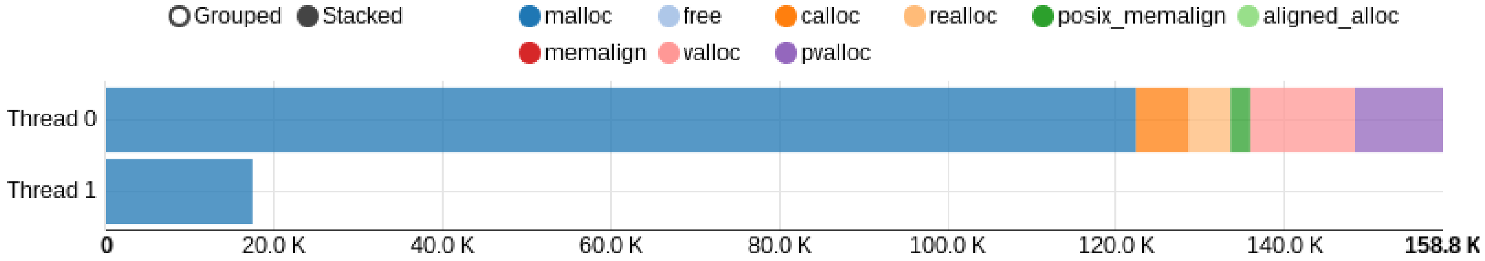
Peak memory

Per thread statistics

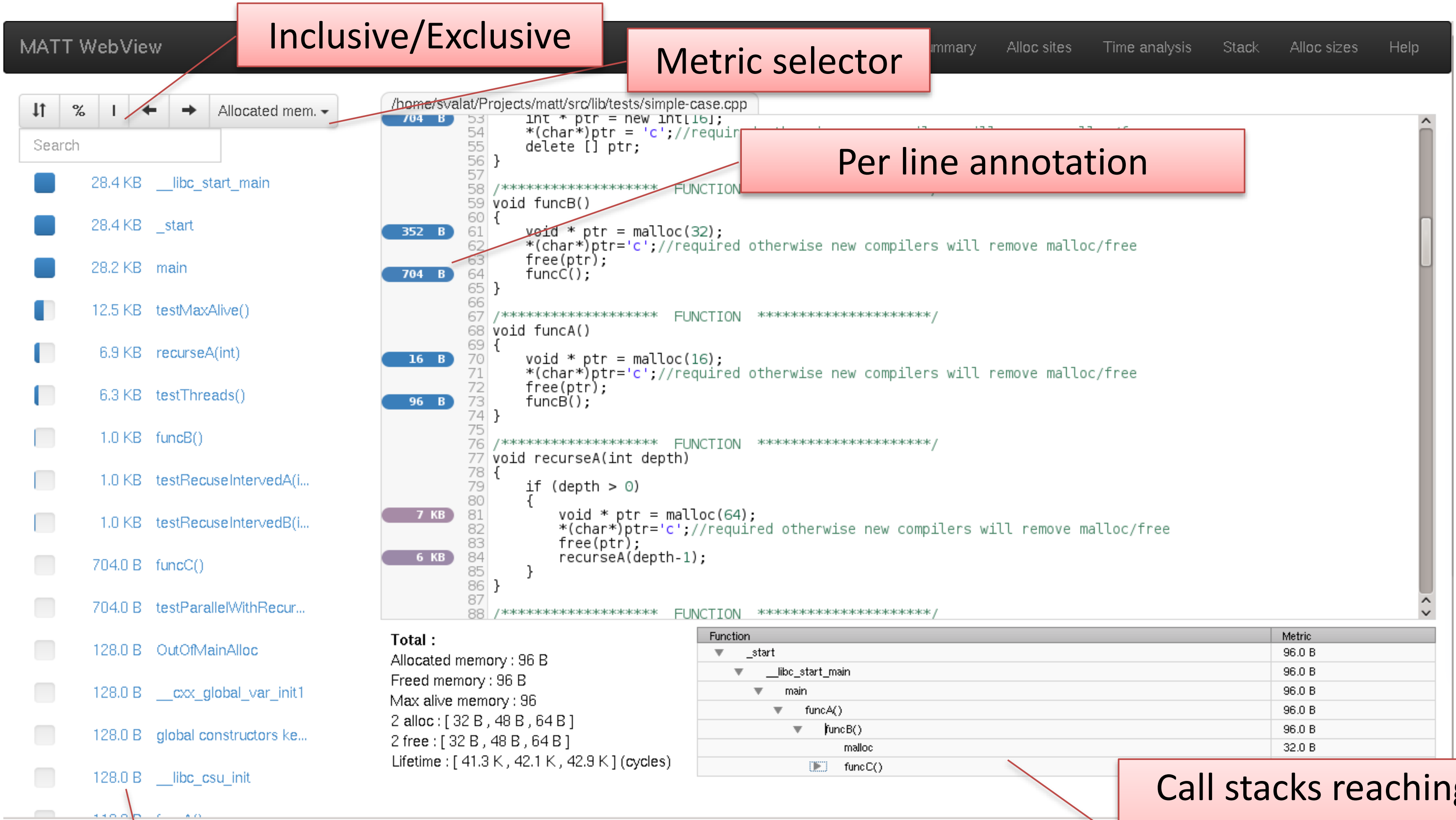
Call per thread



Time per thread



Source annotations



28.4 KB	__libc_start_main
28.4 KB	_start
28.2 KB	main
12.5 KB	testMaxAlive()
6.9 KB	recurseA(int)
6.3 KB	testThreads()
1.0 KB	funcB()
1.0 KB	testRecuseIntervedA(i...
1.0 KB	testRecuseIntervedB(i...
704.0 B	funcC()
704.0 B	testParallelWithRecur...
128.0 B	OutOfMainAlloc
128.0 B	__cxx_global_var_init1
128.0 B	global constructors ke...
128.0 B	__libc_csu_init

/home/svalat/Projects/matt/src/lib/tests/simple-case.cpp

```
    int * ptr = new int[16];
    *(char*)ptr = 'c';//requir
    delete [] ptr;
}

/*******************  FUNCTION
void funcB()
{
    void * ptr = malloc(32);
    *(char*)ptr='c';//required otherwise new compilers will remove malloc/free
    free(ptr);
    funcC();
}

/*******************  FUNCTION  *********************/
void funcA()
{
    void * ptr = malloc(16);
    *(char*)ptr='c';//required otherwise new compilers will remove malloc/free
    free(ptr);
    funcB();
}

/*******************  FUNCTION  *********************/
void recurseA(int depth)
{
    if (depth > 0)
    {
        void * ptr = malloc(64);
        *(char*)ptr='c';//required otherwise new compilers will remove malloc/free
        free(ptr);
        recurseA(depth-1);
    }
}

/*******************  FUNCTION  *********************/
```

Total :
Allocated memory : 96 B
Freed memory : 96 B
Max alive memory : 96
2 alloc : [32 B , 48 B , 64 B]
2 free : [32 B , 48 B , 64 B]
Lifetime : [41.3 K , 42.1 K , 42.9 K] (cycles)

Function	Metric
▼ _start	96.0 B
▼ __libc_start_main	96.0 B
▼ main	96.0 B
▼ funcA()	96.0 B
▼ funcB()	96.0 B
malloc	32.0 B
▶ funcC()	

Inclusive/Exclusive

Metric selector

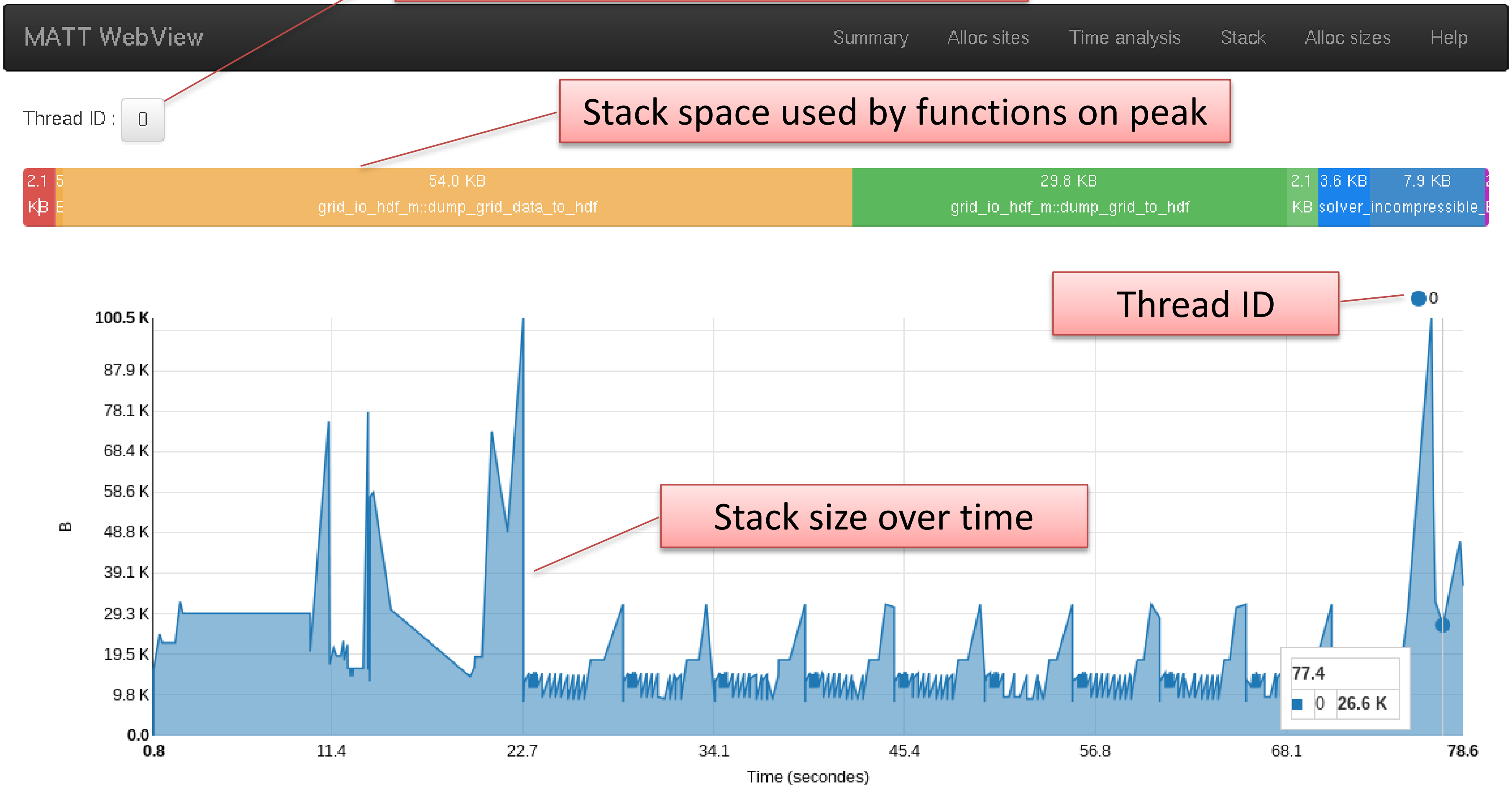
Per line annotation

Call stacks reaching the selected site.

Symbols

Details of symbol or line

Tracking stack memory



Display largest stack for thread ID

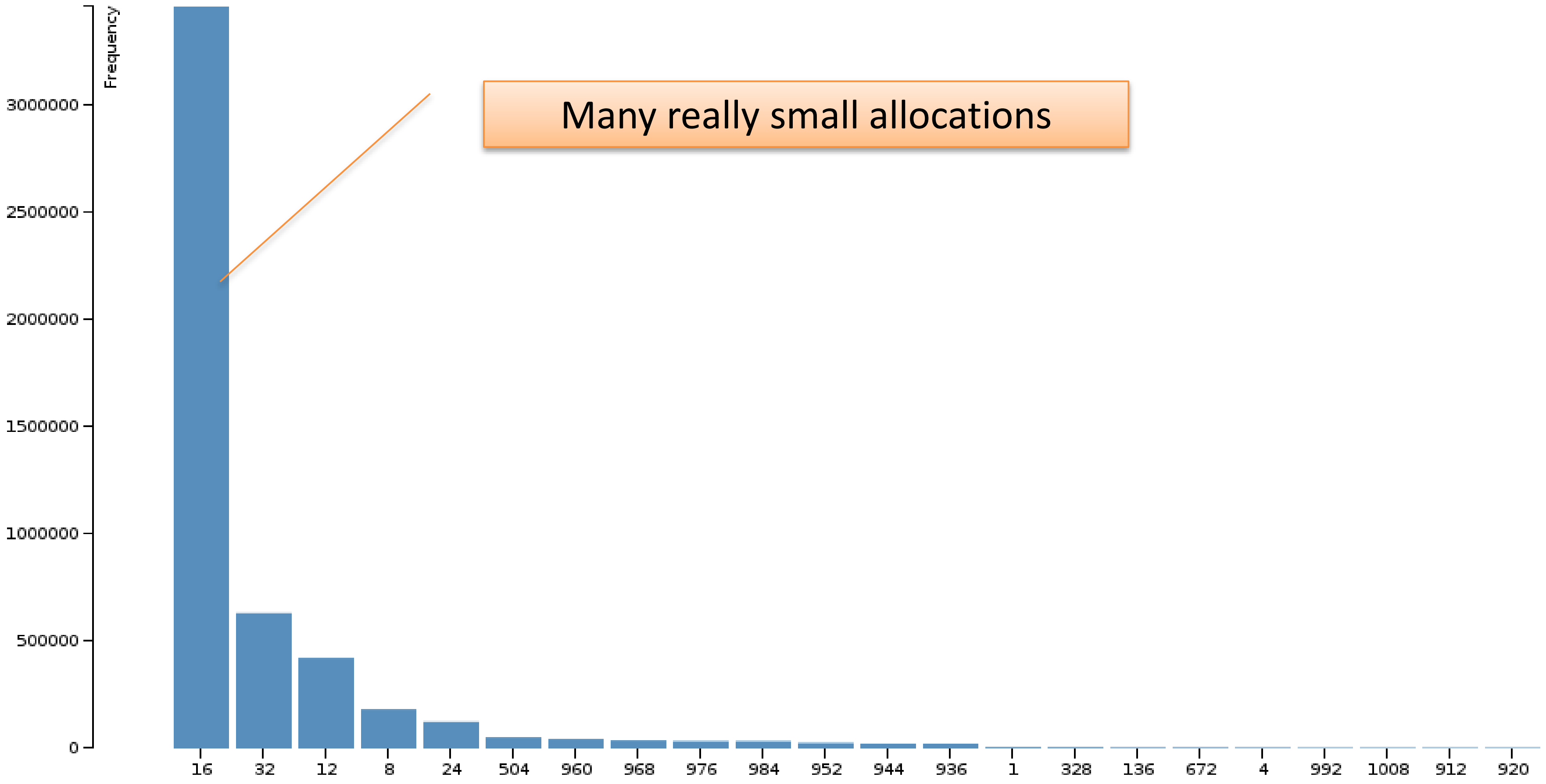
Stack space used by functions on peak

Thread ID

Stack size over time

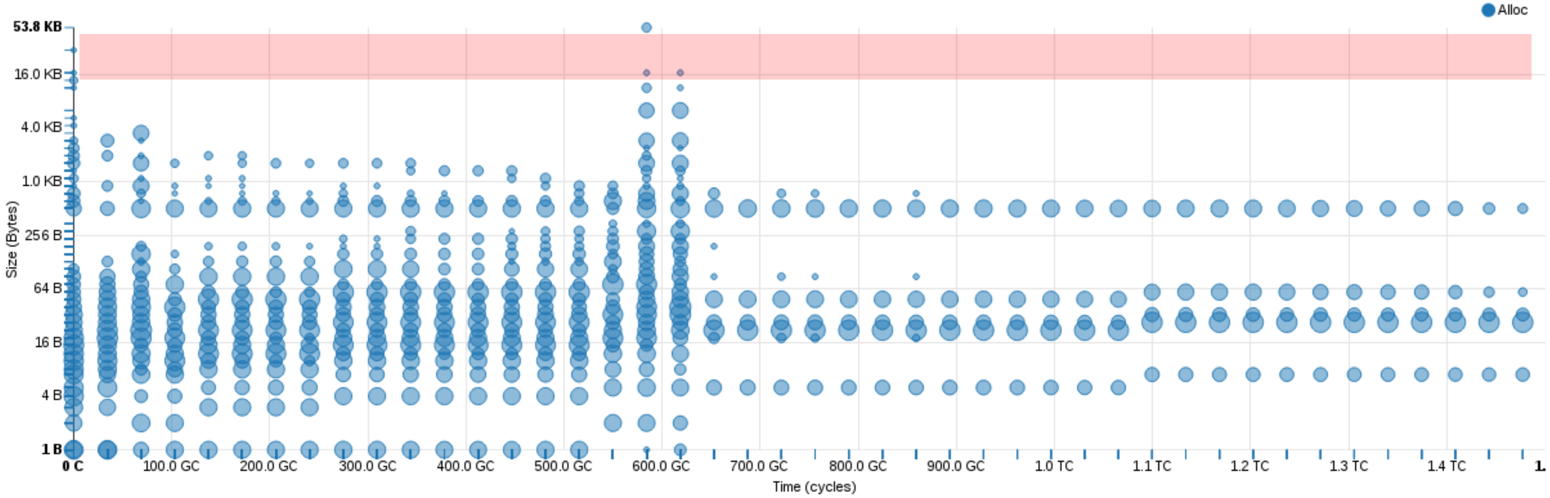
Chunk size distribution

Example from YALES2 with gfortran issue

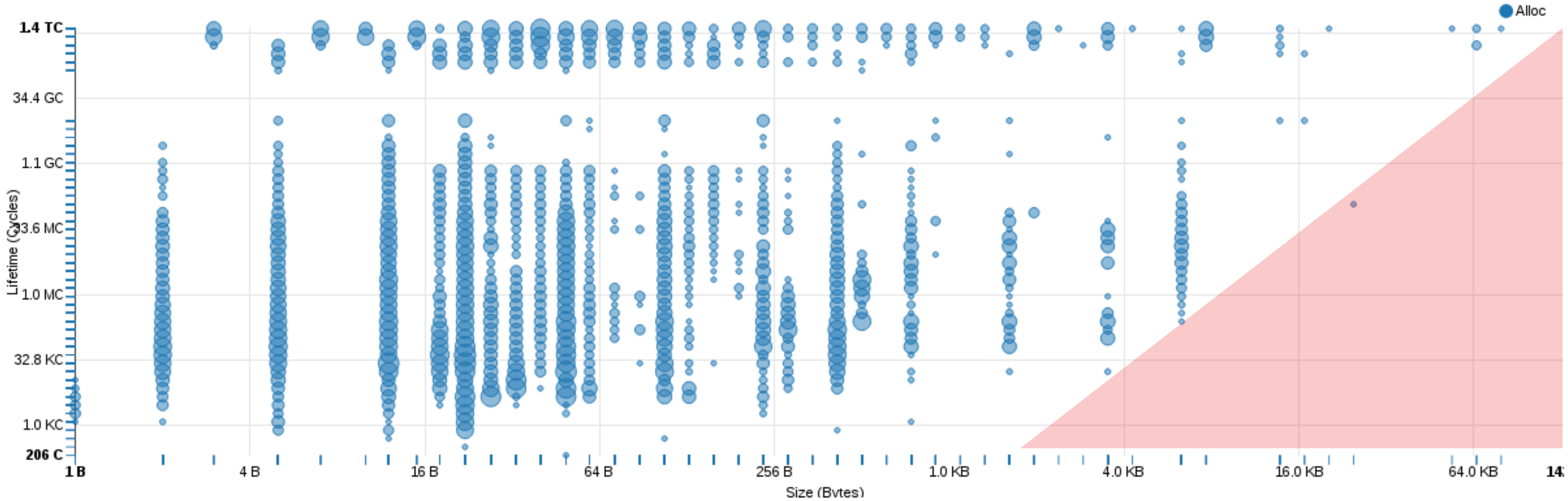


Many really small allocations

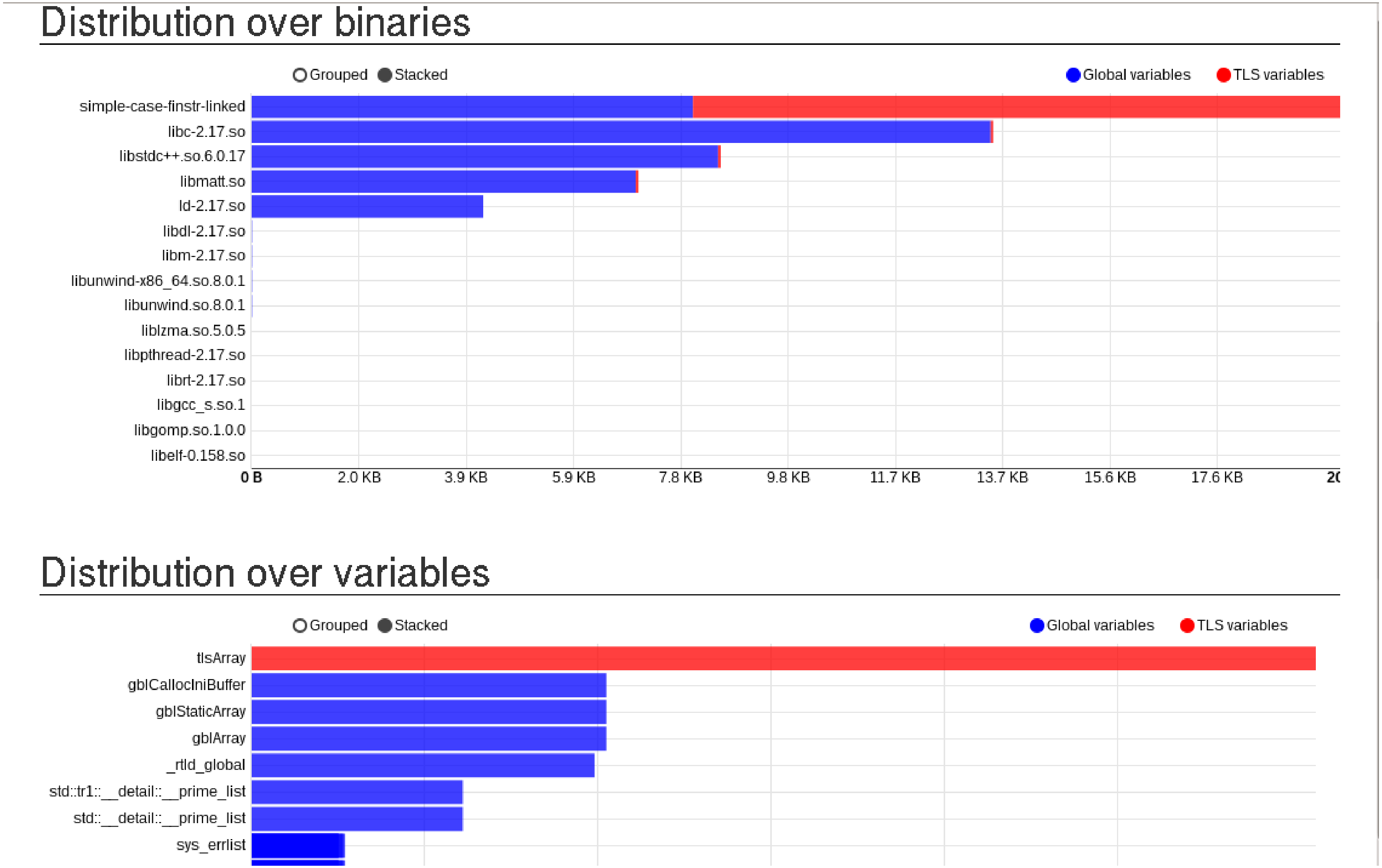
Size over time



Lifetime over size



Global variables



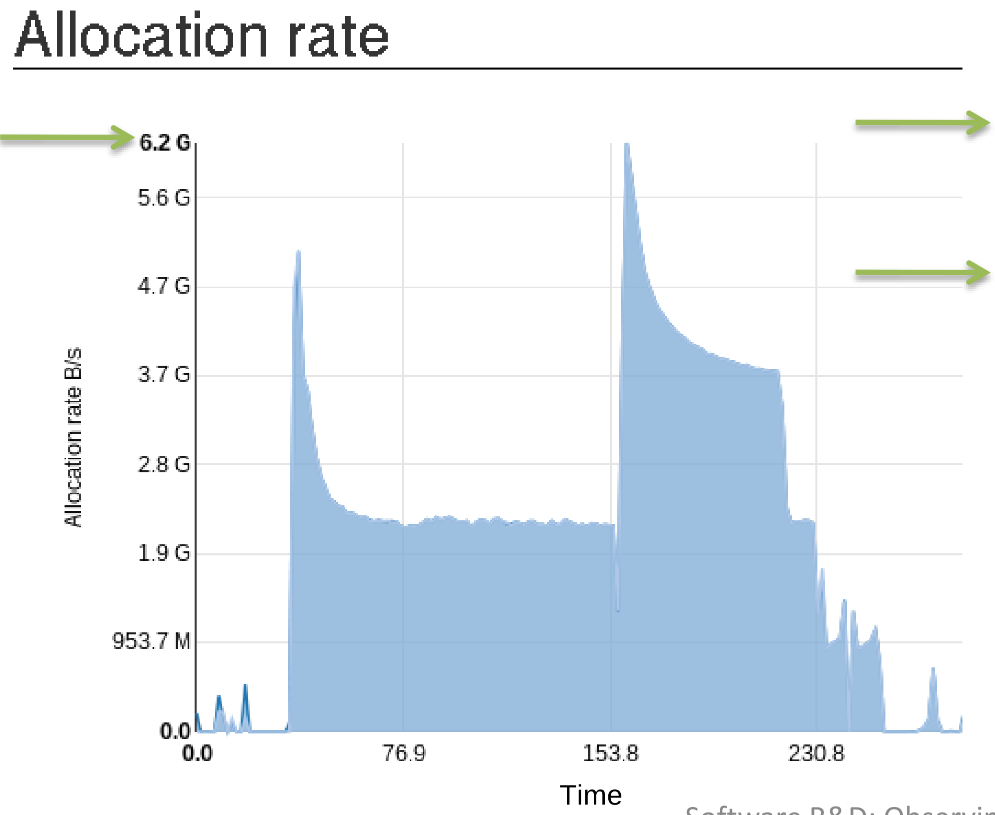
Distribution over binaries

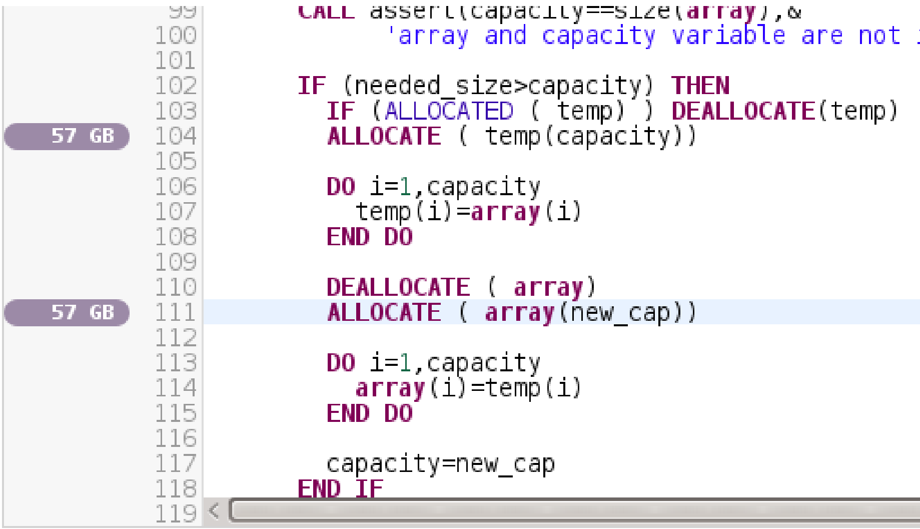
Distribution over variables

REAL CASES

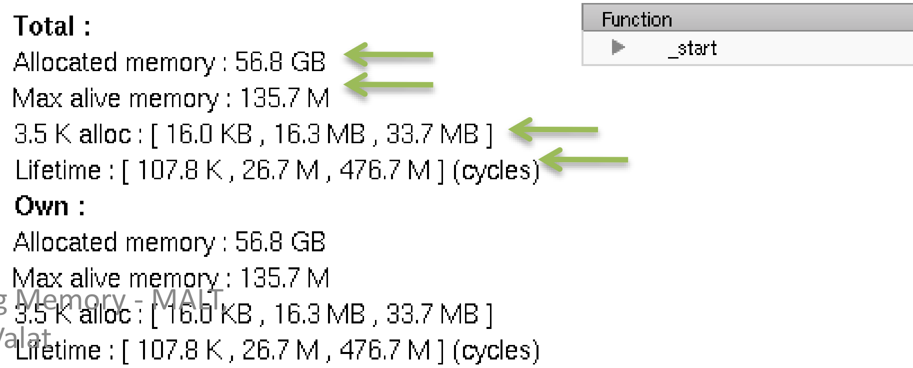
Example on AVBP init phase

- Issue with **reallocation** on init :





```
CALL assert(capacity==size(array),&
      'array and capacity variable are not 

IF (needed_size>capacity) THEN
  IF (ALLOCATED ( temp) ) DEALLOCATE(temp)
  ALLOCATE ( temp(capacity))

  DO i=1,capacity
    temp(i)=array(i)
  END DO

  DEALLOCATE ( array)
  ALLOCATE ( array(new_cap))

  DO i=1,capacity
    array(i)=temp(i)
  END DO

  capacity=new_cap
END IF
```



Total :

Allocated memory : 56.8 GB

Max alive memory : 135.7 M

3.5 K alloc : [16.0 KB , 16.3 MB , 33.7 MB]

Lifetime : [107.8 K , 26.7 M , 476.7 M] (cycles)

Own :

Allocated memory : 56.8 GB

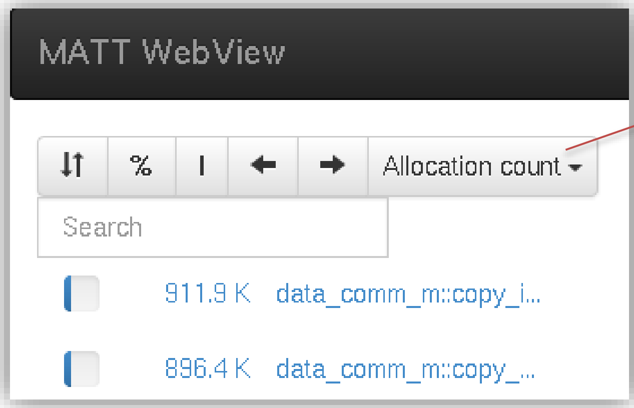
Max alive memory : 135.7 M

3.5 K alloc : [16.0 KB , 16.3 MB , 33.7 MB]

Lifetime : [107.8 K , 26.7 M , 476.7 M] (cycles)

Allocatable arrays on YALES2

- Issue only occur with **gfortran**, ifort uses stack arrays.



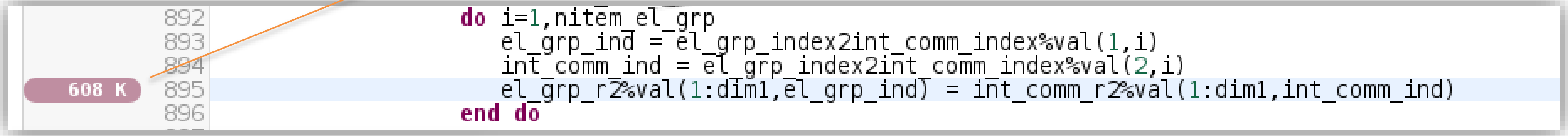
MATT WebView

911.9 K data_comm_m::copy_i...

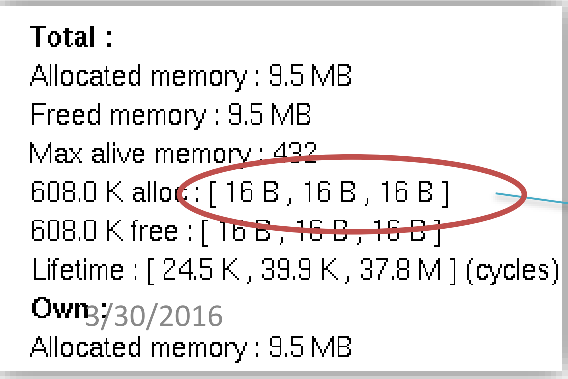
896.4 K data_comm_m::copy_...

Search intensive alloc functions

Huge number of allocation for a line programmer think it doesn't do any !



```
            do i=1,nitem_el_grp
               el_grp_ind = el_grp_index2int_comm_index%val(1,i)
               int_comm_ind = el_grp_index2int_comm_index%val(2,i)
608 K          el_grp_r2%val(1:dim1,el_grp_ind) = int_comm_r2%val(1:dim1,int_comm_ind)
            end do
```

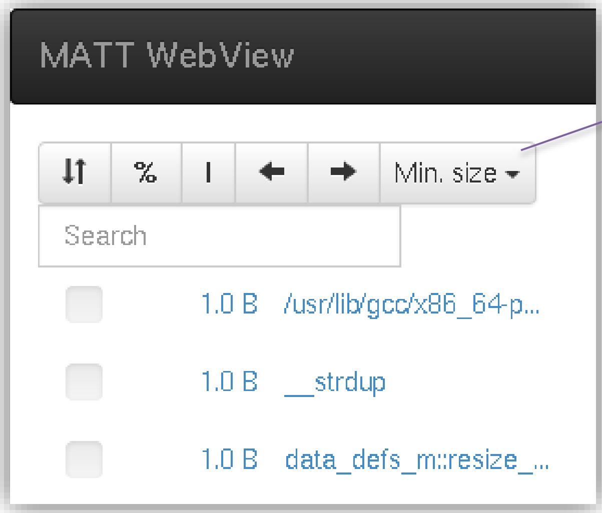


Total :
Allocated memory : 9.5 MB
Freed memory : 9.5 MB
Max alive memory : 432
608.0 K alloc : [16 B , 16 B , 16 B]
608.0 K free : [16 B , 16 B , 16 B]
Lifetime : [24.5 K , 39.9 K , 37.8 M] (cycles)
Own :
Allocated memory : 9.5 MB

And mostly really small allocations !

We can found allocs of 1B !

- Examples on YALES 2, small allocations :

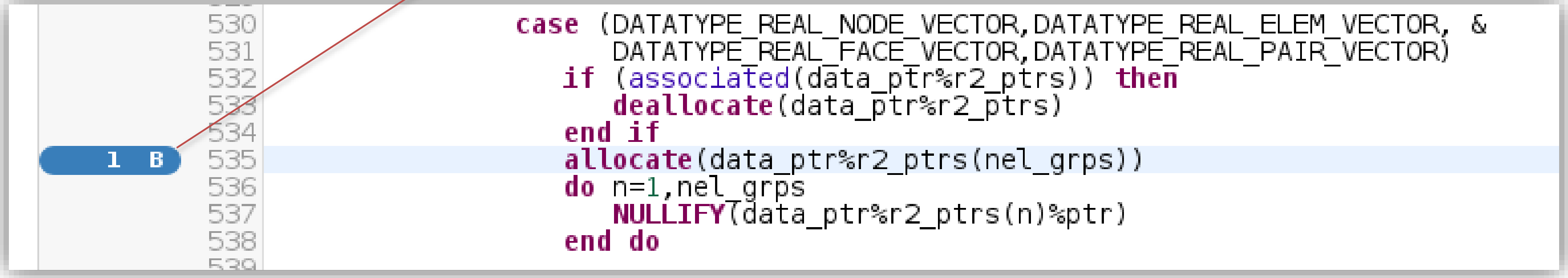


MATT WebView

□	1.0 B	/usr/lib/gcc/x86_64-p...
□	1.0 B	__strdup
□	1.0 B	data_defs_m::resize_...

Search for the minimal chunk size.

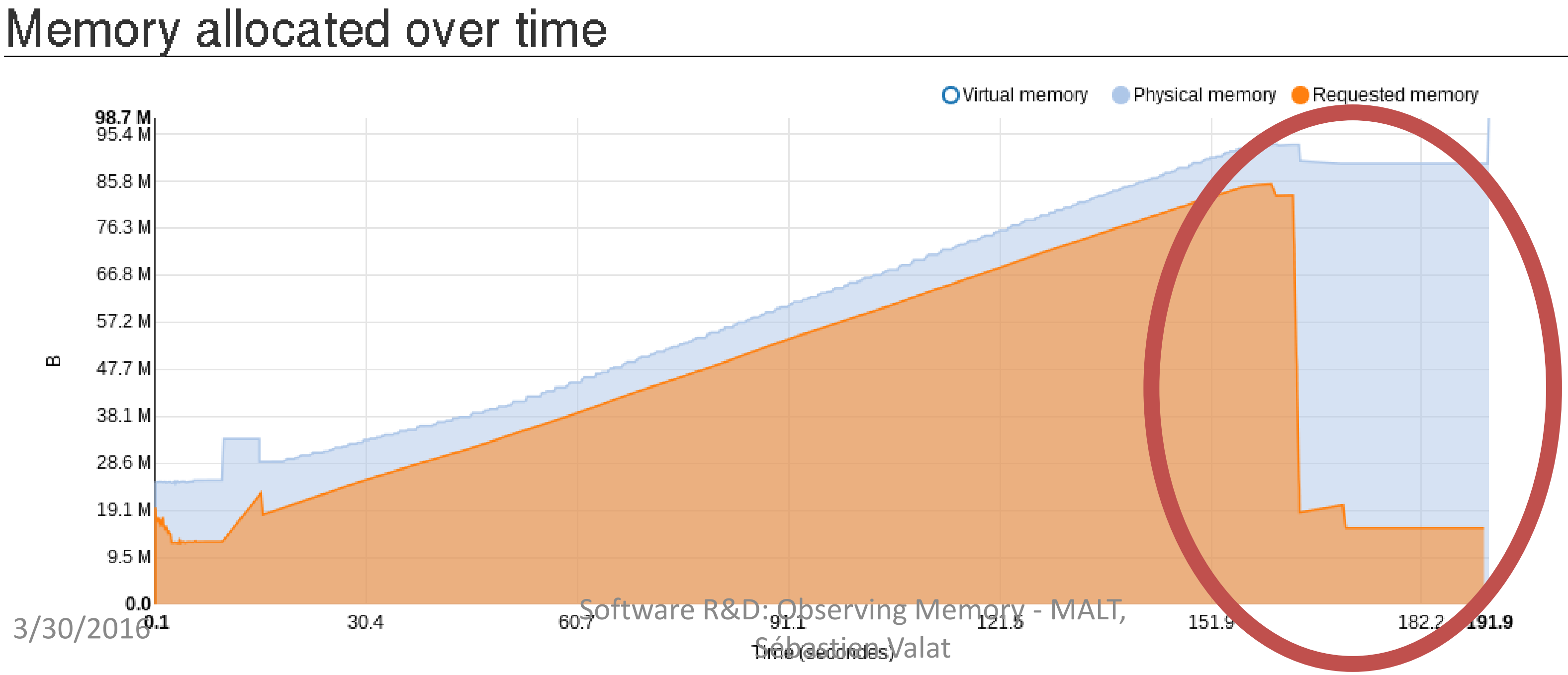
Many codes produce allocations of 1B.
OK with moderation.



```
case (DATATYPE_REAL_NODE_VECTOR,DATATYPE_REAL_ELEM_VECTOR, &
      DATATYPE_REAL_FACE_VECTOR,DATATYPE_REAL_PAIR_VECTOR)
   if (associated(data_ptr%r2_ptrs)) then
      deallocate(data_ptr%r2_ptrs)
   end if
   allocate(data_ptr%r2_ptrs(nel_grps))
   do n=1,nel_grps
      NULLIFY(data_ptr%r2_ptrs(n)%ptr)
   end do
```

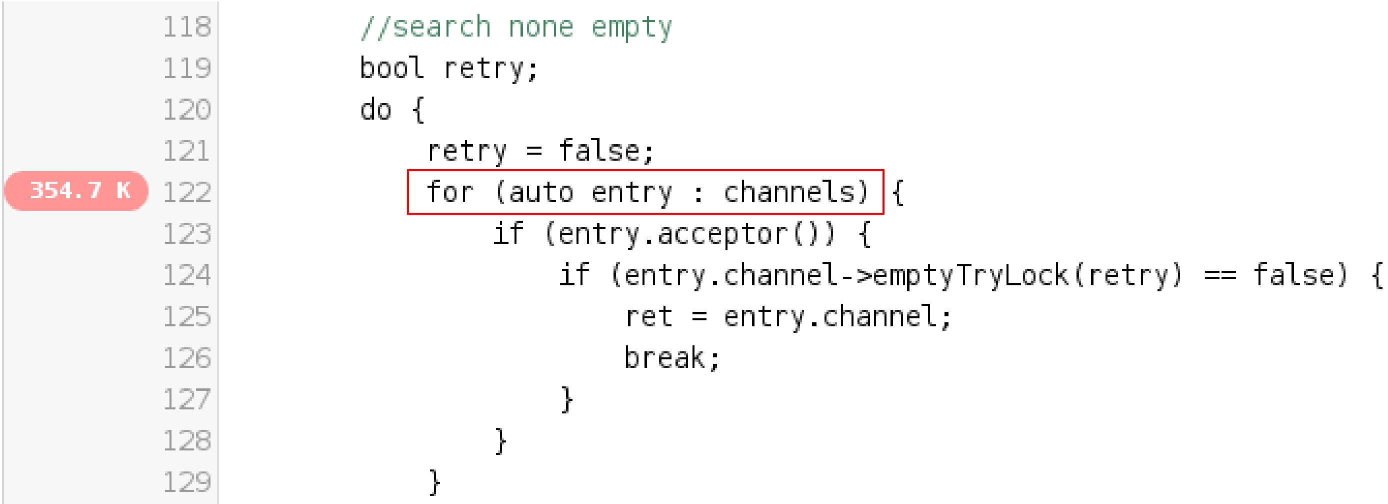
Fragmentation issue

- Example from **Dassault mini-app** from Loïc Thébault and Eric Petit.
- **Fragmentation** can **prevent** from **returning physical pages** to OS
- **Solution** : **avoid interleaved** allocation of chunks with **different lifetime**.
- We observed with the **source annotation** that **most of them can be avoided**.


Memory allocated over time

Missing reference on auto

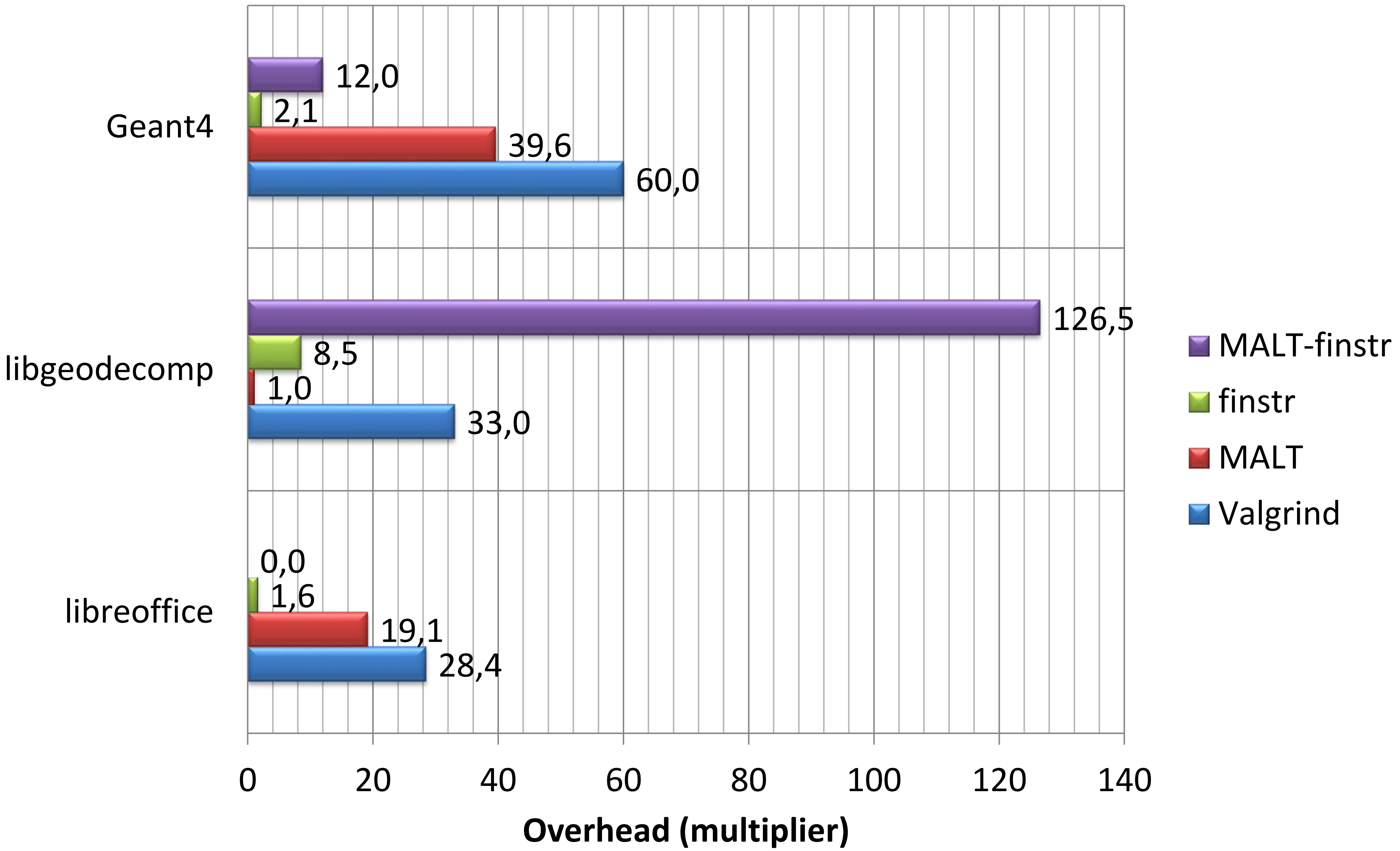
- On one of my own code:



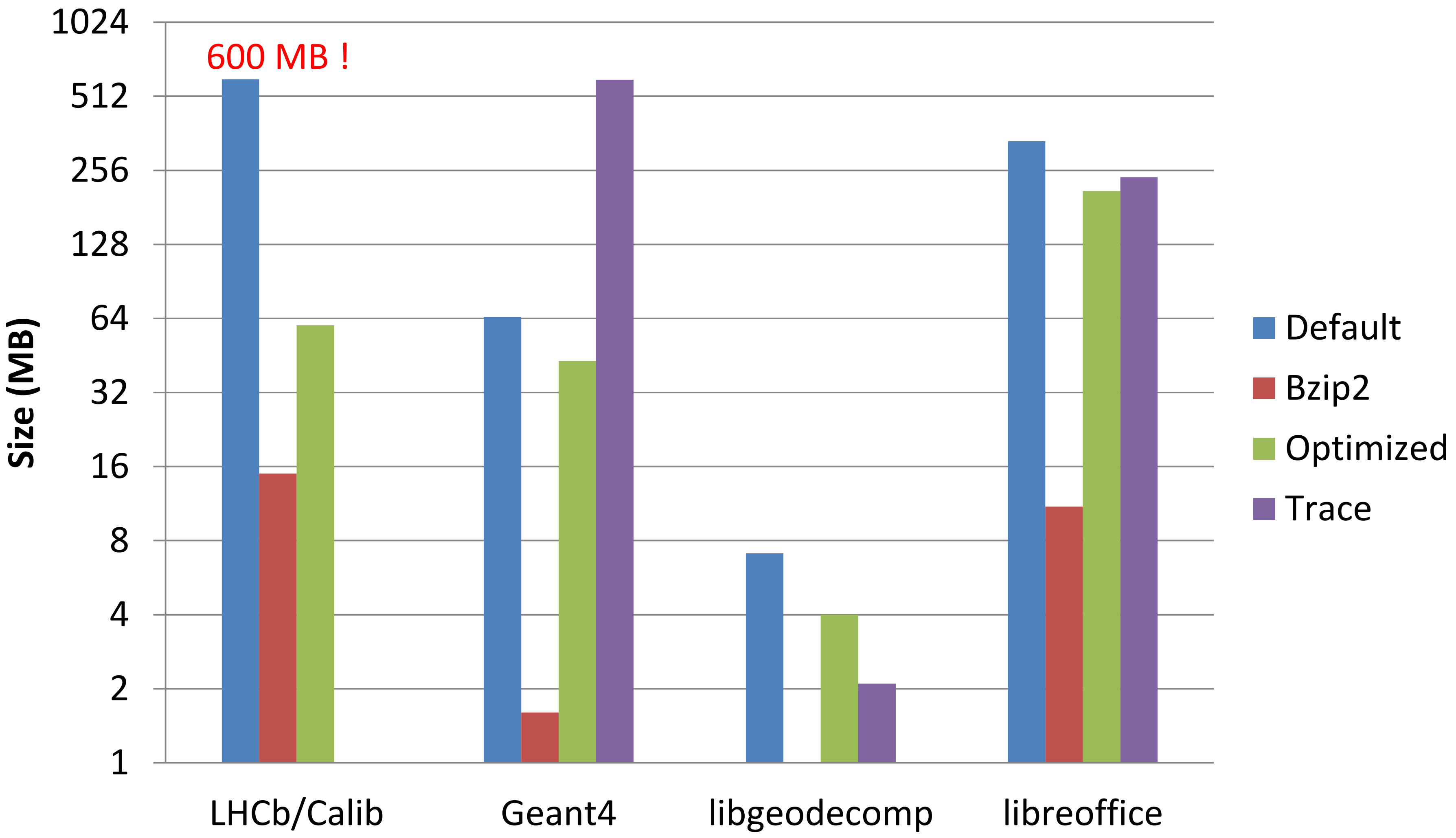
```
//search none empty
bool retry;
do {
    retry = false;
    for (auto entry : channels) {
        if (entry.acceptor()) {
            if (entry.channel->emptyTryLock(retry) == false) {
                ret = entry.channel;
                break;
            }
        }
    }
```

- It was in a **active waiting loop**….
- **Divide by 300** the number of allocation

Overhead and profile size



Profile file size



USAGE & CONCLUSION

Usage

- Backtrace mode :

```
# Optionally recompile with debug flag  to get source lines :
cc –g …
# Run your program
${PREFIX}/bin/malt [--config=file.ini] YOUR_PRGM [OPTIONS]
```

- Function tracking with -finstrument-function :

```
# Recompile with instrumentation flag :
cc -finstrument-function –g …
# Run
${PREFIX}/bin/malt --stack=enter-exit [--config=file.ini] YOUR_PRGM [OPTIONS]
```

- Use the web view :

```
#Launch the server
malt-webserver -i  malt-{YOUR_PRGM}-{PID}.json
# Connect with your browser on http://localhost:8080
```

Conclusion

- Useful tool for tracking allocation patterns

- Interesting real cases :
  - YALES2 : allocations with gfortran
  - AVBP : large allocation rate
  - PAMPA : allocation larger than expected by programmer
  - Dassault mini-app : effect and source of fragmentation

- Future work :
  - Integrate traces into the view (already get all the backend stuff)
  - Add NUMA informations (at lease statistics about mappings)
  - Hope to get Open Source release soon

Thank you.

QUESTIONS ?

BACKUP

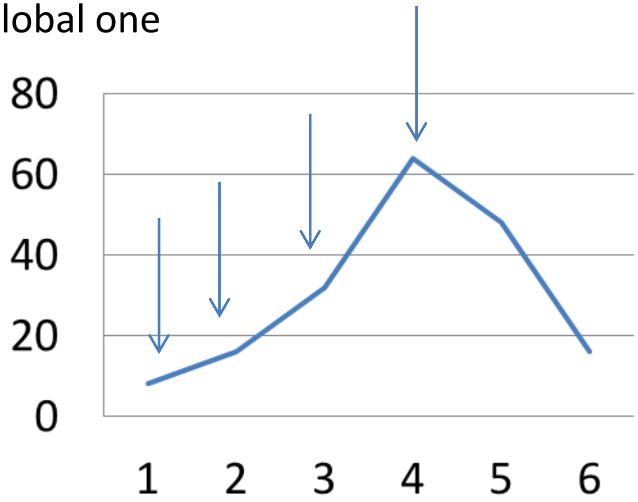
Ideas of improvement

- Add NUMA statistics
- Provide virtual/physical ratio
- Estimate page fault costs
- Exploit traces in GUI for deeper analysis
  - Alive allocations at a certain time
  - Fragmentation analysis
  - Time charts from call sites
  - Usage over threads for call sites

About global peak

- The tool maintain a **call stack tree**
- Profile **stats on leafs**
- On **new** global **peak**, need **to copy** each **local current contribution**
- Need to **walk over** the wall **tree** each time ?

- Do **lazy update** :
  - Keep track of **last local peakId** on each leaf
  - On leaf update, compare the **local peakId** and the global one
  - If not same : remember the old local contribution


80
60
40
20
0
1
2
3
4
5
6